רב-מכר

מהדורה שלישית

C++ בקלות

ייעוץ מקצועי: אבי בוץ







הוצאת הוד-עמי
לספרי מחשבים

C++ בקלות

מהדורה שלישית

קרא את קובץ ONCD.DOC בתקליטור כדי לקבל מידע על תכולת התקליטור המצורף.

קרא בהקדמה כיצד להתקין את המהדר TCLite וכיצד להעתיק את קוד המקור.

עורך ראשי: **יצחק עמיהוד**

תרגום לעברית: **שמואל בן-טולילה, ירון ותמי תלם**

עריכה מקצועית של המהדורה השלישית: **אבי בוץ, חברת מיי-סופט**

עריכה ועיצוב: **ענבל אילני, שרה עמיהוד**

עיצוב עטיפה: **שרון רז**

לשם שטף הקריאה כתוב ספר זה בלשון זכר בלבד. ספר זה מיועד לגברים ונשים כאחד ואין בכוונתנו להפלות או לפגוע בציבור המשתמשים/ות.

- **טלפון: 09-9564716**
- **פקס: 09-9571582**
- **דואר אלקטרוני: Info@hod-ami.co.il**
- **אתר באינטרנט: http://www.hod-ami.co.il**

C++ בקלות

מהדורה שלישית

ד"ר קריס ג'מסה

עורך מקצועי של המהדורה:

אבי בוץ, מיי-סופט





הוצאת הוד-עמי
לספרי מחשבים

Rescued by C++

By Kris, Jamsa, Ph.D.

Third Edition

Editor: **I. Amihud**

הוצאת הוד-עמי
לספרי מחשבים בע"מ
ת.ד. 6108 הרצליה 46160
טלפון: 09-9564716 פקס: 09-9571582

הודפס בישראל

1/2000

HOD-AMI Ltd.
P.O.B. 6108, Herzliya
ISRAEL, 1/2000

ISBN 965-361-225-5 מסת"ב

תוכן העניינים

תוכן העניינים

כמה מילים על אבי בוץ

אבי בוץ הינו מומחה לפיתוח מערכות ובסיסי נתונים בסביבת Windows ו-Linux. מזה 10 שנים הוא עוסק בתכנות ובפיתוח תוכנה בשפות ++C/C ו-Visual Basic.

אבי הינו מרצה בכיר במכללת סיון ובמכללות מחשבים מובילות בארץ. אבי הקים את אתר האינטרנט **WWW.C4MATCH.COM**.

דואר אלקטרוני של אבי בוץ : aviboots@netvision.net.il

הקדמה

במה שונה ספר זה מספרים אחרים בנושא?

הספר ***C++ בקלות*** מתחיל **מהבסיס**, משלב קליטת נתונים מהמקלדת, הצגתם על המסך **ועד לנושאים מתקדמים העוסקים בתכנות מונחה עצמים**. המחבר עושה זאת בשיטת **ראה-ועשה**. הוא מסביר ומדגים, ושוב מסביר ושוב מדגים – כך, עד שתרכוש יכולת עבודה עצמית.

בספר תמצא עשרות דוגמאות לבניית תוכניות בשפת C++, תרגילים ופתרונות, אשר יאפשרו לך "לחקות" אותם בעת תרגול, ולבסוף - גם לשפר, להרחיב ולשלב אותם ביישומים שתכתוב. הם ישמשו לך **ערכת כלים** לבניית יישומים של ממש.

זוהי ***מהדורה שלישית*** של הספר, אשר כוללת עדכונים, תיקונים ותוספות, כמתבקש בסביבת המחשוב הדינמית והמתעדכנת.

מבין החידושים של הספר:

- ספריית תבניות סטנדרטיות - STL (פרק 40): מהי וכיצד אפשר להשתמש בה.
- יצירת פרויקט (פרק 41): בניית קוד תוכנית או יישום ב-C++ קרויה "יצירת פרויקט". בפרק זה תלמד מהו מרכיביו של פרויקט וכיצד לערוך אותו ולהביאו למצב של יישום בר-ביצוע.
- הרצת תוכנית חדשה/קיימת ב-Visual C++ (פרק 42): במקרים רבים הצעד הבא בתכנות או בפיתוח מערכות תוכנה הינו מעבר לעולם ה"וויזואלי" של C++ הקרוי Visual C++. בפרק זה תלמד איך להריץ תוכנית C++ שכתבת בעבר או שאתה כותב כעת.
- שיפורים ותיקונים להסברים ולתוכניות, כפי שמצאנו, או כפי שהעירו לנו קוראינו הנאמנים. תודתנו לכולם.

למי מיועד הספר?

הספר ***C++ בקלות*** מיועד למי **שמתחיל את צעדיו הראשונים בתכנות**, וגם **למי שיש לו רקע ומעט ניסיון בתכנות ורוצה לעבור לשפת C++**. גם אם אתה מבין מהם משתנים, לולאות ופונקציות, תמצא תועלת בספר זה. תוכל לעבור במהירות על הפרקים הראשונים ותמצא בו נושאים חדשים המיוחדים לשפת C++. אם אתה יודע

תכנות, אך **תכנות מונחה עצמים - Object Oriented Programming** (**OOP**) לא מוכר לך עדיין - אל דאגה. הספר ***C++ בקלות*** מניח שאין לך רקע קודם בתכנות מונחה עצמים, ועל כן הוא מנחה אותך בשבילים של עולם מופלא ומתקדם זה.

אם אתה מתחיל, מהתחלה... ובכן, זה יהיה מהנה אפילו. C++ היא אכן שפה נפלאה ומורכבת, אך גם העושים את צעדיהם הראשונים בעולם התכנות יוכלו להתמודד איתה בעזרת ספר זה. **הספר שבידך יתן לך יסודות מוצקים בתכנות ברמה הגבוהה ביותר.** תלמד כיצד לתכנת בצורה נכונה, אך, תצטרך לעבוד **לאט**: יהיה עליך לקרוא **בעיון** כל פיסקה ולהריץ כל דוגמה, **לפתור** בעצמך את התרגילים ואחר כך להשוות לפתרון שהספר נותן לך. בסיומו של הספר תדע תכנות, וגם תדע לתכנת יישומים כרצונך. ואז... עבור לשלב הבא בעזרת מיגוון הספרים ש**הוצאת הוד-עמי** מציעה לך.

כיצד להשתמש בספר?

ספר לימוד קוראים מהתחלה. זה נכון, אבל יש עוד מספר דרכים לקריאת ספר זה. לספר יש תוכן עניינים מקוצר, תוכן מפורט מאוד, וגם אינדקס עברי/לועזי. תוכן העניינים המפורט מאפשר לך למצוא במקרים רבים את מבוקשך, ואם לא - יש אינדקס שיביא אותך בדיוק לנושא שחיפשת. לאחר שתעבור ותלמד את הנושאים השונים שבספר, תוכל להניח את הספר בצד ולהיעזר בו מעת לעת לפי הצורך.

מומלץ להקדיש זמן לדפדוף בספר, כדי להכיר אותו בצורה כללית ולהתרשם מיכולות השפה שאותה הינך עומד ללמוד וליישם. כך תדע לאן לפנות כשתיתקל בקושי כלשהו.

מה בספר?

C++ בקלות מחולק לשישה חלקים, בהתאם לשלבי הלימוד של שפת C++. כל חלק עוסק במכלול נושאים המהווים יחידת לימוד אחת:

- **חלק 1** מניח את היסודות לתכנות בשפת C++: משתנים, קלט ופלט בסיסי, משפטי התנייה (if) עם אופרטורים יחסיים ולוגיים, וגם לולאות.
- **חלק 2** עוסק בבניית תוכניות ושימוש בפונקציות, ובכלל זה העברת פרמטרים, שימוש בספריית ההרצה (run-time library), טווח ההכרה של משתנים (variable scope), העמסת פונקציות (overloading functions), משתני ייחוס (C++ references) ודרכים לקביעת ערכי ברירת מחדל לפרמטרים (default parameter values).
- **חלק 3** מציג דרכים לאחסנת נתונים במערכים (arrays) ובמבנים (structures). בחלק זה נלמד גם על עבודה במחרוזות תווים (character strings), על שימוש באיגודים (unions) ועל מצביעים (pointers).
- **חלק 4** מוקדש ללימוד נושא מרכזי בתכנות מונחה עצמים - המחלקות (classes). נלמד גם על נתונים פרטיים (private) וציבוריים (public), על פונקציות בנייה (constructor functions) ופונקציות פירוק (destructor functions). פרק אחד

מוקדש להעמסת אופרטורים (operator overloading) ופרק נוסף – לפונקציות סטטיות (static functions) ומשתני מחלקה (data members).

- ❖ **חלק 5** ממשיך בלימוד תכנות מונחה עצמים ומציג את ההורשה (inheritance) הרגילה וההורשה המרובה, אלמנטים פרטיים וחברים (private members and friends) ואת השימוש בתבניות פונקציה (function templates) ובתבניות מחלקות (class templates).
- ❖ **חלק 6** ממשיך בנושאים מתקדמים של ++C ועוסק בניהול משאבי הזיכרון החופשי (free store), פעולות קלט/פלט בקבצים, ייעול cin ו-cout, שילוב שגרות אסמבלי בתוכניות ++C, ארגומנטים של שורת פקודה, קבועים ומקרוס, פולימורפיזם (polymorphism) וניהול חריגים (exceptions). בנוסף, על STL, יצירת פרויקט וקצת על ++Visual C.

תרגילים לחזרה: בסוף כל פרק תמצא תרגילים אשר עוסקים בנושאים שבהם דן הפרק. פתור אותם בעצמך תחילה, ורק אחר כך קרא מתוך התקליטור את הפתרון המוצע. פתרון זה אינו היחיד וגם אינו בהכרח האופטימלי, או האלגנטי ביותר, אך תוכל בהחלט להשתמש בו כבנקודת התייחסות בלימודיך.

שים ♥ !	**חשוב!** הספר נכתב לצורך לימוד עבודה במהדר בורלנד מתקדם. התרגילים והפתרונות הורצו בגירסה 4 ומעלה של מהדר **++Borland C**. תוכל להריץ אותם גם במהדר אחר, או במהדר TCLite (גירסה קלה של מהדר Borland) שנמצא בתקליטור המצורף. את רוב התוכניות והתרגילים ניתן להריץ במהדר TCLite. **החומר שבפרק 39** מחייב מהדר מגירסה 4.5 ומעלה של בורלנד, או גרסת מהדר אחר ברמה זהה לפחות, או ++Visual C.

בספר זה תמצא כלים שונים שבהם אנו משתמשים לעזר במהלך הלימוד:

הערה 	הערה הקשורה בנושא המדובר.
שים ♥ !	הדגשת עניין בעל חשיבות מיוחדת או אזהרה.

מודגש: טקסט מודגש בעת הופעת מושג בפעם הראשונה ו/או כדי להבליטו בפיסקה.

במסגרת עם הצללה, הסבר נוסף, **בעברית פשוטה** או במילים אחרות, על מושג או נושא שבו דנים בפרק.

תוכנת TCLite בתקליטור המצורף

לנוחותך תמצא בתקליטור מהדר TCLite של בורלנד, באדיבות חברת Borland. באמצעות מהדר זה תוכל להדר את רוב התוכניות. להידור התוכניות המורכבות של C++ תזדקק למהדר שלם של השפה.

מהדר TCLite הוא גירסה קלה של מהדר Borland C++, אשר עוצב על ידי חברת בורלנד, כדי לאפשר תרגול של תוכניות C++.

רוב התוכניות הורצו במהדר זה, אך לא כולן (יש תוכניות שזקוקות למהדר מתקדם יותר). על כן, אם יש קושי בהרצה של תרגילים כלשהם, ראוי שתריץ אותם לבדיקה במהדר שלם.

הוראות ההתקנה וההפעלה המלאות נמצאות בתקליטור בקובץ Software\tclite\install.doc

אם מותקן כבר מהדר Borland במחשב, אל תתקין את TCLite אלא רק העתק את התוכניות מהתקליטור.

התיקיה \software\tclite

תוכנת **TCLite** (Turbo C Lite), מהדר לשפת C\C++ של חברת Borland International, אשר מיועד למחשבי IBM PC ותואמים. זהו מהדר מהיר מאוד המסוגל, לפי דיווחי החברה, להדר מעל 7,000 שורות בדקה.

התוכנה כוללת:

- מהדר (compiler).
- עורך (editor) המשמש לכתיבת ותיקון התוכניות.
- מערכת תפריטים נוחה, לבחירת האופציות השונות במערכת.
- מקשר (linker).
- מערכת איתור שגיאות בהידור.
- מערכת לבניית תוכניות מקבצי מקור רבים.

הודעה

רשיון השימוש לתוכנה מוגבל להדגמה בלבד. התוכנה משווקת כמו שהיא ("**as is**"), ללא אחריות לתוצאות העלולות להיגרם מהשימוש ו/או אי השימוש בתוכנה מכל סיבה שהיא, וכן להפסד כספי, זמן, רכוש ו/או כל הפסד אחר שייגרם, אם ייגרם.

התקנת התוכנה

ההתקנה וההפעלה פשוטים. עקוב אחר הסעיפים הבאים. ניתן להתקין תחת Windows אך להפעיל, רצוי וצריך תחת **DOS**.

ההתקנה תבוצע בסביבת העבודה DOS, ולצורך ההסבר נתייחס לכונן :C (אותן פעולות תעבודנה גם על כל כונן אחר).

>>אם אתה נמצא ב-Windows - כבה והפעל את המחשב במצב DOS<<

>>בחלון כיבוי המחשב סמן את האפשרות
>>Restart the computer in MS-DOS mode?

1. יש לעבור לתיקיית השורש על ידי הפקודה: \ cd (קיצור של change directory), ולהגיע למצב זה: _<\:C.

2. במקרה וזה איננו ניסיון התקנה ראשון - יש למחוק את התיקיה TCLITE, שבתוכה תותקן התוכנה. עשה זאת על ידי הפקודה: deltree tclite, ואז תראה על המסך את המשפט הבא: C:\>deltree tclite. לבסוף, הקש <Enter>.

   אם התיקיה לא קיימת, הסמן יעבור מיידית לשורה חדשה וימתין להוראה. אם התיקיה קיימת, תופיע השאלה הזו:

   Delete directory "tclite" and all its subdirectories? [yn] _

   כלומר, "האם אתה בטוח שאתה רוצה לבטל תיקיה זו?" הקש על המקש<Y> ואחר כך על <Enter>. המחשב ידווח: ...Deleting tclite.

3. עתה יהיה עליך לעבור לכונן המתאים של התקליטור (למשל :X). המעבר ל- :X - על ידי הפקודה: :x (ואז על המסך יהיה כתוב: C:\>x:). לאחר הקשת <Enter> הסמן המהבהב יופיע אחרי _<\:X.

4. עכשיו הכל מוכן לביצוע ההתקנה. כתוב את הפקודה: X:\>**install c:** (שפירושה, התקנה לתוך כונן :C). שים לב: **אל תשכח** לכתוב :c, כלומר לאן יש לבצע את ההתקנה. המחשב ידווח על התקנת התוכנה TCLITE. משך זמן ההתקנה תלוי במהירות המחשב שלך, ויכול להימשך גם מעל 2 דקות. בדרך כלל ההתקנה מהירה יותר.

5. במקרה של תקלה (תקיעת המערכת) יש לבטל את ההתקנה על ידי <Ctrl>+<C>. במקרה זה המחשב ישאל: ?Terminate batch job (Y/N); כלומר, האם לסיים את העבודה. יהיה עליך להקיש <Y> ו- <Enter>, ואז, בהנחה שיש די זיכרון בדיסק הקשיח, לחזור על שלבים 2-4.

6. לאחר סיום ההתקנה שים את התקליטור בצד ושמור אותו כגיבוי. הפעלת התוכנית והרצת התוכניות נעשית מתוך הדיסק הקשיח.

הפעלת התוכנה

1. יש לחזור מכונן :X לכונן :C, ולהימצא בתיקיית השורש של כונן C.

2. הכניסה ל- ++C - על ידי הפקודה **C:\>c** המפעילה את קובץ ההפעלה C.BAT. כתוצאה, מקבלים את סביבת העבודה של ++Turbo C/C, ובתוכה חלון פתוח בצבע כחול, ששמו: **NONAME00.CPP** (רק בפעם הראשונה שנכנסים לתוכנה). אם הקובץ c.bat לא הועתק לכונן C שלך, העתק אותו ידנית או צור קובץ אצווה חדש. קובץ ההפעלה של התוכנה נמצא ב-tclite\bin\tc.exe.

3. כדאי לסגור את החלון **NONAME00.CPP** על ידי צמד המקשים <Alt>+<F3> ואז מקבלים חלון אפור ריק, שבראשו רשימת תפריטים על רקע לבן. כרגע אתה מוכן להתחיל לעבוד בעזרת התוכניות שבספר.

***שימו לב**: ברירת המחדל של המהדר היא לקרוא ולשמור קבצים עם סיומת C או CPP. בחר c.* או cpp.* בהתאם לקבצים (שפת C או שפת ++C).*

הרצת תוכניות

העתק את התיקיה של התוכניות לפי ההוראות בהמשך.

1. כדי לקבל תוכנית - יש להקיש <F3>. הקשה זו פותחת "חלון שיחה", שכותרתו הראשית: Load a File ("טען קובץ") וכותרת משנה ראשונה הינה: Name ואחריה, בתוך שורה כחולה עם הדגשה ירוקה כתוב: CPP.* (אם אתה מריץ תוכניות C - הקש Backspace פעמיים כדי שיישאר רק c.*. אם אינך רואה דבר הקלד c.*, ולאחר מכן הקש <Enter>).

2. תעבור לחלון פנימי שכותרתו Files ("קבצים"), ותראה על רקע בצבע תכלת 2 טורים של שמות קבצים. באופן כללי, מקשי החיצים מאפשרים לעבור עם ה"סמן", שצבעו ירוק בהיר, על כל הקבצים.

3. בחר את התיקיה הרצויה ואת הקובץ הרצוי. הקשה על <Enter> תכניס אותך לתוכנית. כדי לבחור תיקיה אחרת הקש על ..\ שמופיע בסוף הרשימה של התוכניות.

4. הקשת <Enter> על שם קובץ תפתח חלון כחול, שכותרתו במרכז הינה: "שם הקובץ", ובתוך החלון יופיע הטקסט של תוכנית זו (או של קובץ זה).

5. כדי להריץ את התוכנית עליך להקיש על צמד המקשים: <Ctrl>+<F9>. המחשב יתרגם את התוכנית לשפת מכונה ויבצע אותה. כדי לראות את פלט התוכנית הקש <Alt>+<F5>. יש להקיש על מקש כלשהו כדי לחזור לחלון התוכנית.

6. המשך ההרצה של תוכניות נוספות מומלץ באופן הבא: לסגור תחילה את חלון התוכנית הנוכחית על ידי צמד המקשים <Alt>+<F3>. אחר כך ניתן לחזור על השלבים 1 עד 5, כאשר בשלב 3 אפשר לבחור תוכנית אחרת.

הערה:

בכל שלב בסעיפי הרצת התוכניות אפשר לבטל שלב כלשהו ולחזור לשלב הקודם על ידי המקש <Esc>.

הערה חשובה!!!

אם המחשב שלכם מחובר לרשת, המלצתנו היא - להקיש F5 בזמן אתחול המערכת כדי לדלג על ביצוע הקבצים autoexec.bat ו-config.sys. אם לא תעשו כך, ייתכן שבתוכניות תראו תווים לא מובנים במקום עברית.

קוד המקור בתקליטור המצורף

בכל אחד משִשת חלקי הספר תמצא עשרות תוכניות בשפת ++C. ביניהן תוכניות קטנות ופשוטות ומהן תוכניות ארוכות ומורכבות. אנו מציעים לך ללמוד אותן, וחשוב לא פחות - להריץ אותן במחשב. כדי למנוע ממך את הטרחה, הטעויות ובזבוז הזמן הכרוכים בהבאתן למצב ריצה, התוכניות מצורפות בתקליטור.

התוכניות הורצו ונבדקו, אך ייתכן שתמצא טעות כלשהי בספר או בתוכניות שבתקליטור. אנו מבקשים ממך לבדוק את הנושא בעצמך, כחלק מתהליך הלימוד שלך. נודה לך אם תודיע לנו על השגיאה, אם ישנה, כדי שנוכל לתקן בהדפסת המהדורה הבאה.

בתקליטור תמצא גם את קובץ הפתרונות לתרגילים שבסוף כל פרק (אין פתרונות לכל התרגילים). אנו ממליצים שתפתור את התרגילים בעצמך, טרם שתפנה לפתרון המוצע. זאת ועוד: הפתרון המוצע אינו בהכרח הפתרון היחיד, או הטוב ביותר!

בתיקיה **books\59253** תמצא את קבצי התוכניות בשתי תיקיות:

RESCUED - התוכניות שכלולות בספר.

PITRONOT - הפתרונות לכל התרגילים שבסוף כל פרק.

התיקיה RESCUED

תיקיה זו כוללת 7 תת-תיקיות שבהן תמצא את רוב התוכניות המוצגות בספר. תיקיה זו מחולקת לשבע תת-תיקיות (SEC) המסומנות לפי חלקי הספר (1 עד 6):

SEC1 , SEC2 , SEC3 , SEC4 , SEC5 , SEC6A , SEC6B

(תוכניות חלק 6 נמצאות בשתי תת-תיקיות: לפרקים 31-35 ו- 36-39 בהתאמה).

תוכל לקרוא את הקבצים לעורך ++C של TCLite או לעורך אחר, כרצונך. התוכניות כתובות בקוד מקור של ++C, כך שתוכל לערוך בהן שינויים כרצונך ולהריץ אותן לתרגול. כמה מהן תוכלנה לשרת אותך במסגרת עבודתך.

שמות הקבצים הינם כפי שתמצא במהלך ההסברים והדוגמאות בספר. קבצי ++C נושאים את הסיומת cpp. כמו למשל: first.cpp.

התיקיה PITRONOT

תיקיה זו כוללת את הפתרונות לתרגילים שבכל פרק. כל תוכנית נמצאת בקובץ נפרד, לפי הסימון הבא:

EXnn_xx.CPP

כאשר: nn - מייצג מספר דו-ספרתי של מספר הפרק המתאים.
xx - מייצג מספר סידורי של התרגיל באותו פרק.

לדוגמה:

EX34_2.CPP

פירושו: פתרון לתרגיל 2 שבפרק 34.

העתקת קבצי קוד המקור לדיסק הקשיח

כדי להעתיק את תכולת התיקיה books\59253:

1. לחץ על לחצן התחל, תוכניות, סייר Windows.
2. לחץ על כונן התקליטורים וסמן את התיקיה 59253 שבתיקיה books.
3. גרור אותה לתיקיה Tclite או לתיקיית השורש C.

מכיון שמקור הקבצים הוא התקליטור, הם מסומנים ל**קריאה בלבד**. יש לשנות מאפיין זה כך:

1. בסייר Windows היכנס לתיקיה בדיסק שבה נמצאים הקבצים שהעתקת (c:\tclite\59253 או אחרת).
2. סמן את כולם על ידי Ctrl+A.
3. הצב את סמן העכבר מעל האזור המסומן ולחץ לחיצה ימנית בעכבר.
4. מהתפריט המקוצר בחר **מאפיינים**.
5. בטל את הסימון בתיבה **קריאה בלבד** (דאג שהיא תהיה ריקה).
6. לחץ על **החל**, לחץ **אישור**.

בדוק האם השתנה המאפיין על ידי סימון הקובץ: לחיצה ימנית ובחירה ב**מאפיינים** בתפריט המקוצר. שים לב שהתיבה של **קריאה בלבד** תהיה ריקה.

התוכניות והפתרונות המצורפים הורצו ונבדקו. עשינו את מירב המאמצים כדי לבדוק את תקינות פעולתן, ולא נותר לנו אלא לאחל לך תרגול מהנה ומועיל.

שים לב, שאם אתה עובד במהדר TCLite - הכל מתבצע בסביבת המהדר. לא יוצרים קובץ EXE ולא יוצאים ל-DOS כדי להריץ אותו. הספר נכתב לשימוש במהדר רגיל ולכן יש לשים לב לשינויים כאשר עובדים ב-TCLite.

חלק 1

נושאים בסיסיים

בחלק הראשון של הספר נציג את הנושאים הבסיסיים הדרושים לבניית תוכניות בשפת ++C.

חלק זה כולל את עקרונות התכנות מהצעד הראשון, ומעניק לקורא את הידע הבסיסי הדרוש לכתיבה ותכנות בשפת ++C.

חלק זה כולל את הפרקים הבאים:

- ❑ פרק 1 - התוכנית הראשונה
- ❑ פרק 2 - מבט על שפת התכנות ++C
- ❑ פרק 3 - כתיבת הודעות על המסך
- ❑ פרק 4 - משתנים לאחסון המידע
- ❑ פרק 5 - פעולות אריתמטיות בסיסיות
- ❑ פרק 6 - קליטת נתונים מהמקלדת
- ❑ פרק 7 - הרצת תוכנית על פי תנאים
- ❑ פרק 8 - לולאות

פרק 1

התוכנית הראשונה

השימוש במחשב הולך ומתפשט. מספר רב של תוכנות משרתות סוגים שונים של משימות בעבודה היומיומית. בין התוכנות הנפוצות ניתן למנות מעבדי תמלילים, גליונות אלקטרוניים וסביבת עבודה DOS, חלונות - Windows או UNIX. למעשה, תוכניות מחשב, או תוכנה (software) הן קבצים המכילים רצף של פקודות הקובעות כיצד המחשב יפעל. קבצים שהסיומת שלהם com או exe הם תוכניות הרצה המכילות את הפקודות להפעלת המחשב, ואשר פועלות במחשבים אישיים.

במילים אחרות, הקבצים מכילים פקודות מסוימות שהשילוב והמבנה שלהן מאפשרים את ביצוע המשימה הרצויה. בעת כתיבת התוכניות, המתכנת מציין רצף פקודות לביצוע פעולות במחשב. בפרק זה נציג כיצד לכתוב פקודות אלו בשפת ++C.

בהמשך הפרק נדון בנושאים הבאים:

- כדי לכתוב תוכנית יש להשתמש **בעורך** (text editor) להקלדת פקודות ++C שיוצרות את קובץ המקור.
- כדי להפוך תוכניות ++C לתוכניות הניתנות להרצה (מורכבות מקוד בינארי, הכולל את הספרות 0 ו- 1 בלבד, ושאותו המחשב מבין), משתמשים בתוכנית ייעודית הנקראת מהדר ++C.
- כדי לשנות את קוד התוכנית משתמשים בעורך.
- כאשר עוברים על כלל תכנות אחד או יותר של ++C, המהדר מציג על המסך הודעת שגיאת תחביר מתאימה (Syntax error). במקרה זה יש לערוך שוב את התוכנית, לתקן את השגיאות שנפלו בה ואז – להדרה שנית.

ניתן להגדיר את עבודת התכנות כתהליך יצירה של רצף הפקודות שהמחשב יריץ כדי לבצע משימה מסוימת ומוגדרת. הכלים ליצירת רצפים אלה הם **שפות התכנות** (**Programming language**), ביניהן שפת ++C. המתכנת כותב את רצף הפקודות **בקובץ מקור** (**Source file**), המכיל את רשימת הפקודות בשפת התכנות שבחר. הדרך המוצלחת ביותר להבנת עבודת התכנות היא על ידי התנסות וכתיבה של תוכניות בשפת ++C.

יצירת התוכנית הראשונה

כעת ניצור את התוכנית הראשונה בשפת ++C. לקובץ המכיל את התוכנית ניתן את השם **FIRST.CPP**. הסיומת cpp מציינת שהתוכנית שבקובץ כתובה בשפת ++C. המשימה שעל התוכנית לבצע, פשוטה: הצגת המשפט "!++Rescued by C" על המסך. בשורות הבאות מוצגים **סימן ההנחיה** (**Prompt**) של מערכת הפעלה DOS (בדוגמה <\:C), הפקודה להרצת התוכנית והודעת הפלט שהיא מציגה על המסך:

```
C:\>FIRST  <Enter>
Rescued by C++!
```

כידוע, את מלאכת התכנות ניתן לבצע ישירות ברמת שורת הפקודה (בסביבה כגון MS-DOS או UNIX), או לחלופין בסביבה המבוססת על Windows. למען הפשטות, כאשר נציג את פלטי תוכניות הדוגמה המופיעות בספר זה, נצא מנקודת הנחה כי קוראינו עובדים ישירות דרך שורת הפקודה. במקרה זה, לדוגמה, כדי להריץ את התוכנית FIRST.EXE צריך להקליד תחילה את שם התוכנית FIRST בשורת הפקודה של המערכת, ולאחר מכן להקיש על Enter.

שים לב, שאם אתה עובד במהדר TCLite - הכל מתבצע בסביבת המהדר. לא יוצרים קובץ EXE ולא יוצאים ל-DOS כדי להריץ אותו. הספר נכתב לשימוש במהדר רגיל ולכן יש לשים לב לשינויים כאשר עובדים ב-TCLite.

השלב הראשון של כתיבת התוכנית היא עריכת קובץ מקור שלה באמצעות עורך, כמו edit (המסופק עם חבילת מערכת ההפעלה DOS). חשוב להדגיש, כי אין להשתמש במעבדי תמלילים כגון Word, מכיון שקבצים הנערכים באמצעות תוכנות אלה הינם בעלי פורמט מיוחד הכולל התייחסות לצורה החיצונית של המסמך כמו: אותיות מודגשות, יישור פסקאות, קביעת שוליים ותכונות נוספות. מעבדי התמלילים משלבים תווים מיוחדים בתוך הטקסט, שלפעמים נסתרים למשתמשים, ואשר מטרתם לסמן את הצורה החיצונית של הטקסט. פענוח הסימנים נעשה על ידי תוכנת מעבד התמלילים שיצרה אותם. לסימנים אלה אין משמעות בשפת ++C, והימצאותם בתוך התוכנית עלולה לגרום לתקלות בתכנות. המהדרים של ++C "מבינים" קבצי ASCII "נקיים".

בשלב ראשוני של הלימוד מומלץ לטעון תוכנת עורך (או במהדר TCLite) ולכתוב את שורות התוכנית המופיעות להלן:

```
#include <iostream.h>

void main(void)
 {
    cout << "Rescued by C++!";
 }
```

נלמד את פירוש הפקודות, או משפטי התוכנית (program statements) האלה, בפרק 2. בשלב זה נעסוק בכתיבת הפקודות בלבד. יש לשים לב לתחביר: דיוק ברישום

ומיקום הפסיקים, נקודות-פסיק והסוגריים המסולסלים. לאחר בדיקה חוזרת ניתן לשמור את רצף הפקודות בקובץ ששמו **FIRST.CPP**.

משמעות השם של התוכנית

קבצים המכילים תוכניות ++C מסתיימים בסיומת (Extention) זו - **CPP**. השימוש במוסכמה זו מקל עלינו ועל מתכנתים אחרים להבחין כי קובץ מסוים מכיל תוכנית ++C. מומלץ ביותר לתת לתוכניות שמות ברורים, המעידים על תפקידן. לדוגמה שם של תוכנית המטפלת בנושאי תקציב, יהיה BUDGET.CPP. באופן דומה, לתוכנית שתפקידה הוא חישוב משכורות כדאי לקרוא SALARY.CPP.

הערה


כדי למנוע בלבול, לעולם אין לתת לתוכנית שם של פקודת MS-DOS, כמו למשל, COPY או DEL.

הידור התוכנית - קומפילציה

המחשב פועל על פי שפה מיוחדת, **שפת מכונה** (**Machine language**), שמתבססת על צירוף של הספרות 1 ו- 0. משמעות הספרות היא נוכחות או היעדר של אות חשמלי. לכן, לפני הרצת תוכנית המקור, הכתובה בשפת התכנות, צריך להמיר אותה לשפת מכונה. ההמרה מתבצעת על ידי תוכנית מיוחדת – **מהדר** (**Compiler**) והפעולה נקראת **הידור** (**Compilation**). לכל שפת תכנות יש מהדרים מסחריים שונים. הנה הפקודה הדרושה להידור של תוכנית FIRST במהדר ++C של חברת Borland (לא ב-TCLite):

```
C:\>BCC FIRST.CPP <Enter>
```

הערה חשובה: אם אתה משתמש במהדר המצורף בתקליטור - TCLite, גרסת המהדר הזו אינה כוללת את הפקודה BCC. את ההידור יש לבצע מתוך המהדר. פעל לפי ההוראות בהקדמה ובקובץ install.doc.

במילים אחרות, המהדר עובר על קובץ המקור המכיל את תוכנית ++C ומנתח אותו. אם משפטי התוכנית אינם מפרים אף כלל תכנות של ++C, המהדר ממיר את הקוד לשפת מכונה (ספרות 0 ו- 1) שהמחשב מבין ויודע לבצע. הסיומת של קובץ המכונה הנוצר היא EXE, קיצור של Executable. כעת, כשקובץ המכונה מוכן, ניתן להריץ את התוכנית על ידי הקלדת שמה בשורת הפקודה, כמקובל.

צורת ההפעלה של מהדר ++C תלויה בסוג המהדר שברשותך. לדוגמה, אם אתה מפעיל את המהדר ++BORLAND C, הידור התוכנית **FIRST.CPP** למשל, יתבצע באמצעות הפקודה BCC כמו בדוגמה זו:

```
C:\>BCC FIRST.CPP <Enter>
```

הוראות להידור באמצעות מהדרים של חברות אחרות מופיעות בספרות הנלווית למהדר. לאחר סיום ההידור נוצר קובץ הרצה. בסביבת העבודה של מערכת הפעלה DOS, תוכנית הרצה תקבל את שם קובץ המקור והסיומת שלו EXE. בדוגמה שלנו הקובץ יהיה **FIRST.EXE**.

הערה חשובה: למרות שהמהדר TCLite המצורף בתקליטור יוצר תוכנית EXE - לא ניתן להריץ אותה משורת הפקודה (זו המשמעות של גרסת Lite לעומת הגירסה המלאה). פעל לפי ההוראות בהקדמה ובקובץ install.doc.

ייתכן כי בשלב ההידור תופענה הודעות שגיאה על המסך. במקרה זה יש לטעון את קובץ המקור בעורך התוכניות, ולבדוק את תוכנו מול הרשום בספר. לאחר תיקון הטעויות יש להדר את הקובץ שנית. לאחר שההידור מסתיים בהצלחה, צריך להריץ את התוכנית על ידי הקלדת שמה בשורת הפקודה, כפי שהוסבר.

המהדר - מה הוא עושה?

כדי ליצור תוכנית, משתמשים בפקודות של שפת תכנות מסוימת (++C לדוגמה). הפקודות הדרושות נכתבות בקובץ מקור. בהמשך יש להפעיל תוכנית מיוחדת הקרויה מהדר (Compiler), הממירה את קובץ המקור לשפת מכונה (שהיא השפה שהמחשב מבין). אם ההידור הסתיים בהצלחה, נוצר קובץ הרצה. לעומת זאת, אם בקובץ המקור נכתבו הפקודות לא לפי התחביר והחוקים של השפה - המהדר יציג על המסך הודעות שגיאה. המתכנת צריך לתקן את השגיאות בקובץ מקור, ולהדר שוב את התוכנית.

אם אתה עובד על מחשב מרכזי או על מיני-מחשב, ייתכן וכבר מותקן עליו מהדר הנגיש לך ולכל שאר משתמשי המערכת. אך אם אתה עובד על מחשב אישי, יהיה עליך לרכוש ולהתקין מהדר מסחרי, כגון ++Borland C או ++Microsoft Visual C. ללימוד בלבד, ואולי קצת יותר, תוכל להסתפק במהדר TCLITE, שנמצא בתקליטור המצורף.

יצירת התוכנית השנייה

כעת נעבור לתוכנית השנייה. לקובץ מקור של התוכנית ניתן את השם **EASY.CPP**. בתוכו נכתוב את הפקודות המופיעות כאן:

```
#include <iostream.h>

void main(void)
 {
    cout << "Programming in C++ is easy!";
 }
```

לאחר שמירת הקובץ, נהדר אותו ונשתמש בשם הקובץ, כדי להפעיל את התוכנית. הפקודה הדרושה לביצוע הידור של תוכנית EASY במהדר שפת C++:

```
C:\>BCC EASY.CPP  <Enter>
```

סיום מוצלח של ההידור ייצור קובץ הרצה לתוכנית, ששמו **EASY.EXE**. כאשר מריצים את התוכנית, יוצגו על המסך השורות הבאות:

```
C:\>EASY   <Enter>
Programming in C++ is easy!
```

כעת נשנה בקובץ המקור **EASY.CPP**, נערוך אותו מחדש ונוסיף את המילה **very** במקום המתאים בהודעת הקלט של התוכנית, כפי שמוצג כאן:

```
cout << "Programming in C++ is very easy!";
```

נשמור את הקובץ ונהדר שוב. בהרצה זו התוכנית תציג את ההודעה החדשה:

```
C:\>EASY  <Enter>
Programming in C++ is very easy!
```

כל פעם שעורכים שינויים בקובץ מקור, יש להדר את התוכנית שוב, כדי שהשינויים יהיו תקפים בהרצת התוכנית.

כל שינוי בתוכנית מחייב את הידורה מחדש, כדי שהשינויים יבואו לידי ביטוי בקוד שפת המכונה. למשל, השתמש בעורך כדי לשנות את קובץ המקור, אך הפעם, הוסף שורה, כפי שתוכל לראות בדוגמה הבאה:

```
#include  <iostream.h>

void main(void)
{
    cout << "Programming in C++ is very easy!";
    cout << endl << "And pretty cool!";
}
```

את השינויים יש לשמור בקובץ המקור. כעת ניתן להריץ את התוכנית, וגם נקבל את התוצאות בשורה שתחת פקודת ההרצה:

```
C:\>EASY <Enter>
Programming in C++ is very easy!
```

כפי שניתן לראות, לא הוצגה שורת הפלט החדשה, אשר נוספה בדוגמה זו. כאמור, יש להדר את התוכנית מחדש כדי שהשינויים בקובץ המקור ייכנסו לתוקף. לפיכך, **נהדר** עתה את התוכנית פעם נוספת. כעת, כאשר נריץ את התוכנית, יופיע על המסך הפלט במלואו, כפי שרצינו:

```
C:\>EASY <Enter>
Programming in C++ is very easy!
And pretty cool!
```

להזכירך: אם אתה משתמש במהדר המצורף בתקליטור - TCLite, גרסת המהדר הזו אינה כוללת את הפקודה BCC. את ההידור יש לבצע מתוך המהדר, ולמרות שהמהדר TCLite המצורף בתקליטור יוצר תוכנית EXE - לא ניתן להריץ אותה משורת הפקודה (זו המשמעות של גרסת Lite לעומת הגירסה המלאה). פעל לפי ההוראות בהקדמה ובקובץ install.doc.

שגיאות תחביר, מהן?

בכל שפה ושפה קיימים חוקים לכתיבת משפטים. חוקים אלה הם **תחביר השפה** (**Syntax**). לדוגמה, בעברית המשפטים מסתיימים בנקודה והסימן לחלוקת המשפט לקטעים הוא הפסיק. בשפות לועזיות כגון אנגלית, צרפתית או ספרדית מתחילים משפט באות גדולה. בדומה לשפת אנוש, גם לשפות תכנות יש תחביר. בשפת ++C יש שימוש בסימן נקודה-פסיק, בסוגריים מסולסלים ובסימנים אחרים על פי התחביר שלה. בביצוע הידור של תוכנית, המהדר מגלה את שגיאות התחביר שבתוכנית המקור ומציין סוג השגיאה ומספר השורה שבה מופיעה שגיאה. המהדר ייצור תוכנית הרצה, רק אם בתוכנית אין שגיאות תחביר.

כדי להמחיש מהן שגיאות תחביר ניצור את התוכנית הבאה:

```
#include <iostream.h>

void main(void)
 {
    cout << Use quotes around messages;
 }
```

בהשוואה לתוכניות הקודמות, בתוכנית האחרונה הודעת הפלט אינה מופיעה בתוך גרשיים. תחביר בשפת התכנות ++C דורש כי הודעות פלט יופיעו בין גרשיים ולכן, בביצוע הידור יוצגו על המסך שגיאות תחביר שהתגלו. לדוגמה, הודעות שגיאה לפי מהדר ++Borland C:

```
C:\>BCC SYNTAX.CPP  <Enter>
Borland C++  Version 4.0 Copyright (c) 1993 Borland
International syntax.cpp:
Error syntax.cpp 5: Undefined symbol 'Use' in function
main()
Error syntax.cpp 5: Statement missing ; in function main()
*** 2 errors in Compile ***
```

במקרה זה המהדר מציג שתי הודעות על שגיאות תחביר, שתיהן בשורה חמש של קובץ המקור. השלב הבא הוא לערוך את הקובץ ולרשום גרשיים בתחילה ובסיום של הודעת הפלט, כפי שמוצג להלן:

```
cout << "Use quotes around messages";
```

לאחר תיקון השגיאה, ניתן להדר את התוכנית וליצור קובץ הרצה. בדרך כלל, בתחילת הדרך בתכנות, מתגלות שגיאות תחביר רבות, אך כשצוברים ניסיון, מספר שגיאות התחביר יורד ומהירות הגילוי שלהן עולה.

שגיאות תחביר, מהן ?

תוכנית בשפת תכנות ++C חייבת להיכתב לפי חוקי השפה, שנקראים תחביר השפה. לדוגמה, הודעות פלט צריכות להיכתב בין גרשיים ובסוף כל פקודה חייב להופיע נקודה-פסיק. אם בתוכנית קיימות שגיאות תחביר, יוצגו על מסך בעת ההידור הודעות מתאימות. המהדר איננו מסוגל ליצור קובץ הרצה לתוכנית שיש בה שגיאות תחביר, ולכן יש "לנקות" את קובץ המקור מכל שגיאת תחביר, כדי שנוכל להמשיך ביצירת התוכנית.

עבודה בסביבת Windows

כדי לפשט את הדוגמאות שהובאו עד עתה, יצאנו מנקודת הנחה כי סביבת העבודה שלך מבוססת על שורת פקודה, כמקובל במערכות ההפעלה MS-DOS או UNIX. למרות זאת, רוב מתכנתי ++C עובדים כיום בסביבת עבודה המבוססת על Windows עם מהדר כמו ++Visual C, או בסביבת הפיתוח המשולבת של Borland.

כאשר מתכנתים בסביבת Windows, משפטי התוכנית אינם שונים מהמשפטים בהם משתמשים בסביבת עבודה המבוססת על שורות פקודה. השוני בין שתי סביבות העבודה הוא בעיקר בתהליך ההידור וההרצה של התוכנית.

בתכנות בסביבת Windows, בה ניתן ליצור קובץ מקור על ידי שימוש בעורך פנימי של המהדר. אחר כך ניתן להדר את התוכנית על ידי בחירה באופציה הרצויה מתוך התפריט, או על ידי לחיצה בעכבר (לחצן שמאלי, כמובן). שגיאות תחביר עשויות להיות מוצגות בחלון נוסף. בסיום ההידור אפשר להריץ את התוכנית על ידי בחירת אפשרות הרצה מתוך התפריט (או בסמל שעל סרגל הכלים).

סביבת העבודה, או **סביבת התכנות** (Programming environment) נקראת כך, מכיון שהיא מספקת את כל הכלים ליצירה, הידור והרצה של תוכניות.

סיכום

בפרק זה למדנו כיצד ליצור ולהדר תוכניות בשפת ++C.

בפרק הבא נציג את החלקים השונים של תוכניות בשפת ++C. נסביר את השימוש בסוגריים המסולסלים { } ובמילות מפתח כגון void. נלמד גם כיצד לכתוב תוכנית המציגה הודעת פלט על המסך.

לפני שנעבור לפרק הבא, מומלץ לבחון אם מובנים הנושאים הבאים:

- ✓ תוכנית היא רצף הפקודות שהמחשב מסוגל לבצע.
- ✓ תוכניות כתובות בשפת תכנות ++C מאוחסנות בקבצים בעלי סיומת CPP.
- ✓ המהדר ממיר תוכניות כתובות בשפת ++C לשפת מכונה, שהיא השפה שהמחשב מבין.
- ✓ לשפת ++C קיים קובץ חוקים, הנקרא תחביר בשפה.
- ✓ כאשר בתוכנית מופיעה שגיאת תחביר, המהדר מציג הודעה המתארת את השגיאה.
- ✓ המהדר ייצור קובץ הרצה לתוכניות, רק כשאין בהן שגיאות תחביר.

תרגילים

1. כיתבו תוכנית המדפיסה את שמכם על המסך. למשל, עבור השם David, הפלט שיתקבל על המסך יהיה:

```
My name is David
```

2. בצעו בתוכנית הקודמת את השינוי הבא, כדי שהיא תדפיס את השורה:

```
My name is David and I'm learning C++
```

הערה: *אל תשכחו להפעיל את המהדר על קובץ הקוד המעודכן!*

3. התוכנית הבאה מכילה שגיאות תחביריות. הפעילו את מהדר ++C ותקנו את התוכנית על פי הודעות השגיאה שהתקבלו.

```
#include <iostream.h>

void main(void)
  {
      cout << The program contains bugs for you to fix! ;
  }
```

פרק 2

מבט על שפת התכנות ++C

בפרק הקודם כתבנו מספר תוכניות בשפת ++C. מטרתנו בפרק הקודם הייתה להציג את התהליך הכתיבה וההידור של תוכניות ולא עסקנו בהבנת פקודות השפה. כפי שראינו, רוב תוכניות בשפת ++C שומרות על מבנה דומה. מתחילות במספר משפטֵי include#, בהמשך מופיע משפט קבוע void main(void) וסדרה של משפטי ביצוע בתוך סוגריים מסולסלים { }. בפרק זה נסקור את המשפטים שהוצגו בפרק הקודם, ונבין את משמעותם. בהמשך הפרק נדון בנושאים הבאים:

- משפט include# מאפשר להשתמש בקבצי כותר המכילים תוכניות, הגדרות והצהרות.
- ביצוע של תוכנית ++C מתחיל מהמשפט **void main(void)**.
- תוכניות מורכבות מפונקציות (אחת או יותר), הבנויות מסדרות משפטים המבצעים יחד מטלה מסוימת.
- הצגת פלט על המסך מתבצעת דרך ערוץ הפלט cout.

תוכנית ++C מורכבת מסוגים שונים של משפטים ופקודות. בפרקים הבאים נציג דוגמאות שונות: **משפט השמה** (**assignment statement**),שבאמצעותו מציבים ערכים בתוך משתנים, או **משפט התנאי if,** המאפשר לתוכנית לקבל החלטות ביצוע בעת ההרצה על פי תנאים שקבענו מראש. בשלב זה נציג באופן כללי **משפטי השפה**, או **משפטי התוכנית** (**program statements**).

משפטי התוכנית

בפרק 1 יצרנו תוכנית ++C בשם **FIRST.CPP**, קובץ המקור של התוכנית הכיל את המשפטים שלהלן:

```
#include <iostream.h>

void main(void)
 {
   cout << "Rescued by C++!";
 }
```

במקרה זה, התוכנית מכילה שלושה משפטים. הסוגריים המסולסלים תוחמים משפטים שיש ביניהם קשר, או רצף, בביצוע בתוכנית. לכן מכנים אותם גם בשם **סמלי הקבצה** (Grouping symbols).

```
#include <iostream.h>

void main(void)
 {
   cout << "Rescued by C++!";
 }
```

משפטי תוכנית

בהמשך נלמד כל אחד מהמשפטים המופיעים בתוכנית.

משפט include#

כל אחת משלוש התוכניות שיצרנו בפרק 1 מתחילה במשפט include# הבא:

```
include <iostream.h>
```

משפט include# הוא הוראה למהדר להכניס בתחילת התוכנית ולפני ביצוע ההידור את תוכן הקובץ, ששמו מופיע בין הסוגריים. במקרה זה המהדר מתייחס לקובץ iostream.h. האות h היא בדרך כלל הסיומת של שמות הקבצים שמשתלבים בתחילת התוכניות. קבצים אלה הם **קבצי כותר** (Header files). ניתן לראותם כי בספריה המכילה את קבצי המהדר מוגדרת תת-ספריה בשם INCLUDE. בתת-ספריה זו נמצאים מספר קבצי כותר. כל קובץ כותר מכיל הגדרות שונות לביצוע פעולות שונות. לדוגמה, math.h הינו קובץ כותר המטפל בהגדרות ובפעולות מתמטיות. קבצי כותר הם קבצי מלל במבנה ASCII, ולכן ניתן להציג אותם על גבי המסך ולהדפיסם. בשלב זה לא נעסוק בתוכן של קבצים אלה, ורק נסתפק בהבנת השימוש במשפטי include#. בכל תוכנית המוצגת בספר זה מצוין איזה קבצי כותר חייבים להיות מוגדרים בה.

קבצי כותר בשפת ++C

תוכניות ++C מתחילות במשפט include# אחד או יותר, המורים למהדר לכלול בתוכנית את קובץ הכותר (Header file) המוזכר במשפט. בסיום ההידור, התוכנית כוללת את המשפטים של קובץ הכותר (את התוכן שלו), כאילו נכתבו מראש ישירות בגוף התוכנית. קבצי כותר מסוגים שונים משמשים את התוכנית בביצוע פעולות שונות. לדוגמה, קבצי כותר מסוג מסוים מגדירים פעולות קלט/פלט, וקבצים מסוג אחר מגדירים את השירותים שמספקת מערכת ההפעלה (למשל, הצגת התאריך והשעה ברגע מסוים), וכן הלאה.

כל התוכניות אשר מוצגות בספר זה כוללות בתוכן את קובץ הכותר iostream.h. קבצי כותר, כמו תוכניות ++C עצמן, הם קבצי ASCII, כלומר תווים, אשר ניתן להציג, לקרוא ולהדפיס את תוכנם. כדי להבין טוב יותר את תוכן הקובץ iostream.h, כדאי להתבונן בו באמצעות העורך ולהדפיסו. קובץ זה נמצא בספריה INCLUDE שנמצאת בספריה המכילה את קבצי מהדר ++C.

הערה



לא מומלץ לשנות את תוכן קבצי הכותר אשר מסופקים ישירות עם מהדר ++C, מכיון ששינוי כזה עלול לגרור שגיאות בתוכניות אחרות אשר כוללות אותו, ואינן "יודעות" על השינוי. זכרו, זהו קובץ סטנדרטי! הערה זו אינה חלה, כמובן, על קבצי כותר שכל מתכנת כותב לעצמו.

משפט void main(void)

התוכניות הכתובות בשפת ++C מורכבות ממספר רב של משפטים. סדר הופעת המשפטים אינו חייב להיות הסדר שבו יתבצעו הפקודות בשלב הרצת התוכנית, אולם כל תוכנית ++C מתחילה בפקודה אחת קבועה. המשפט **void main(void)** מאפשר למתכנת לקבוע את המקום שבו תתחיל התוכנית לרוץ. כאשר התוכניות מבצעות פעולות רבות, מספר המשפטים בה גדל, ובהתאם לכך קיים הצורך לפצל את התוכנית לחלקים קטנים יותר, שיעניקו למתכנת שליטה טובה בתוכנית. במקרה זה, המשפט void main(void) מציין את סדרת המשפטים העיקריים (main), שיתבצעו בתחילת ההרצה של התוכנית.

מהי תוכנית ראשית?

קבצי מקור של תוכניות ++C מכילים מספר רב של משפטים. המשפט void main(void) הוא הזיהוי של התוכנית הראשית (Main program), אשר מכילה את הפקודות הראשונות לביצוע. בכל תוכנית ++C חייב להיות משפט אחד בלבד המכיל את השם main.

כאשר בוחנים תוכניות ++C גדולות, כדאי לחפש את המשפט המכיל את השם main. משפט זה הוא למעשה הצהרת הפונקציה ()main, הכוללת את הפקודה הראשונה לביצוע בתוכנית.

הבנת השימוש ב-void

ככל שהתוכניות שנכתבות הולכות ונעשות מורכבות יותר, צריך לחלק אותן לקטעים אשר ניתנים ל״שליטה״, אלו הן **פונקציות** (Function). פונקציה היא קבוצת משפטי תוכנית בתוך התוכנית שמבצעים משימה אחת מוגדרת. לדוגמה, בתוכנית משכורת למשל, הפונקציה salary תחשב את השכר של העובד, והפונקציה printsal תדפיס את תלוש השכר. וכך למשל, הפונקציה squar_root תחשב שורש ריבועי בתוכנית העוסקת בחישובים מתמטיים. כאשר התוכנית מפעילה פונקציה, הפונקציה מבצעת את המטלה ו״מחזירה״ ערך תוצאה אל התוכנית שהפעילה אותה (או ״קראה לה״ - Function call).

בכל תוכנית קיימת פונקציה אחת לפחות, ולכל פונקציה יש שם ייחודי לה. התוכניות שכתבנו בשיעור הראשון כללו אך ורק פונקציה אחת, הפונקציה main. בפרק 9 נחקור לעומק את השימוש בפונקציות. לעת עתה נבהיר, כי פונקציה מורכבת מאוסף משפטים המכוונים לביצוע מטלה מסוימת.

בתוכניות ++C שכיח השימוש במילה void. ניתן להשתמש במילה void כדי לציין כי פונקציה מסוימת אינה מחזירה ערך כלשהו, או כדי לציין כי לא מועברים לפונקציה ארגומנטים בזמן הפעלתה. לדוגמה, תוכניות הפועלות בסביבת MS-DOS או UNIX מסוגלות לסיים את פעולתן בהעברת **ערך סטטוס יציאה** (Exit status value) אל מערכת ההפעלה. את סטטוס היציאה אפשר לבחון למשל, בעזרת קבצי אצווה (Batch files). בסביבת DOS, קבצי אצווה בוחנים את סטטוס היציאה של תוכנית על ידי תנאי IF ERRORLEVEL. בדוגמה הבאה נבחן את סטטוס היציאה של התוכנית **PAYROLL.EXE** המסיימת, ומעבירה אחד מהערכים הבאים למערכת ההפעלה:

ערך יציאה	משמעות
0	הצלחה
1	קובץ איננו קיים
2	אין נייר במדפסת

להלן דוגמה לתוכנית אצווה ב- DOS המתייחסת לתוכנית ולטבלה לעיל:

```
PAYROLL
IF ERRORLEVEL 0 IF NOT ERRORLEVEL 1 GOTO SUCCESSFUL
IF ERRORLEVEL 1 IF NOT ERRORLEVEL 2 GOTO NO_FILE
IF ERRORLEVEL 2 IF NOT ERRORLEVEL 3 GOTO NO_PAPER
REM Other batch file commands here
```

בתוכניות הפשוטות שנציג בספר איננו זקוקים להחזרת קודים למערכת ההפעלה, ולכן נציב את המילה void, לפני שם התוכנית הראשית main כדלקמן:

```
void main(void)
|_______________________ Program does not return a value
```

בפרקים הבאים נציג כיצד ניתן להעביר לתוכנית פרמטרים (שם קובץ קלט, לדוגמה), מתוך פקודת ההרצה של התוכנית. כאשר מריצים תוכנית ללא פרמטרים, קובעים את המילה void בתוך הסוגריים שבהמשך לשם התוכנית הראשית main, כדלקמן:

```
void main(void)
          |
          |_______________________ Program does not use
                                   command line arguments
```

כאשר תוכניות נעשות מורכבות יותר, הן עשויות להחזיר ערכים למערכת ההפעלה או לתמוך בפרמטרים של שורת פקודה. בשלבים אלה נסתפק בהצהרה void עם main, כפי שמופיע בתוכנית.

הסוגריים המסולסלים { }

תוכניות גדולות ניתן לחלק לפי קבוצות של משפטים. בדרך כלל יש גם קבוצה של משפטים שתתבצע מספר רב של פעמים. לדוגמה: פעולת חישוב הציונים של 100 תלמידים. פעולות אחרות שמתקיימות תחת תנאי מסוים, ניתן לרכז בתוך קבוצה של פקודות. לדוגמה, הצגת המילה "עובר" על המסך לתלמיד שקיבל מעל 50 בבחינה. הסוגריים המסולסלים { } מקבצים בתוכם סדרה של משפטים וכך הם מחלקים את התוכנית לקבוצות של משפטים. עד כאן הצגנו את הסוגריים המסולסלים המקבצים את משפטי התוכנית הראשית main.

משפט cout להצגת הודעות פלט על המסך

בתוכניות הקודמות השתמשנו לצורך הצגת הודעות על המסך, השתמשנו במילה cout ובסימן **קטן מ-** כפול (<<), כך:

```
cout << "Hello, C++!";
```

המילה **cout** מסמנת את **ערוץ הפלט** (**Output stream**) ששפת ++C משייכת להתקן הפלט הסטנדרטי המוגדר במערכת ההפעלה. בדרך כלל (אם לא אומרים למערכת משהו אחר), מערכת ההפעלה מנתבת הודעות פלט למסך, מכיון שהוא מוגדר כהתקן הפלט הסטנדרטי.

יחד עם זאת, ניתן לנתב את הפלט ליעדים אחרים כגון קובץ, או התקנים אחרים, כגון מדפסת. בדוגמה הבאה - פלט התוכנית first.exe מנותב למדפסת במקום למסך, על ידי שימוש בפקודה ישירה של מערכת ההפעלה.

```
C:\>FIRST > PRN  <Enter>
```

שים ♥! **קובץ EXE הנוצר תחת TCLite לא יכול לרוץ תחת DOS. הכל מתנהל תחת התוכנית ולא בחלון DOS.**

ערוץ הפלט cout

כפי שלמדנו, תוכניות C++ משתמשות בערוץ הפלט cout כדי להציג הודעות על המסך. הסדר בו שולחת התוכנית תווים אל ערוץ cout הוא הסדר בו הם יופיעו על המסך. למשל, נכתוב את המשפטים הבאים:

```
cout << "This message appears first, ";
cout << "followed by this message.";
```

בתגובה, מערכת ההפעלה תציג את רצף התווים הזה:

```
This message appears first, followed by this message.
```

אופרטור ההכנסה (<<) נקרא כך, מפני שהוא מאפשר לתוכנית להכניס (להזרים) תווים אל ערוץ הפלט.

בפרק 3 נעסוק בשימוש של cout ונציג כיצד ניתן להרכיב משפט פלט הכולל טקסט, מספרים שלמים ומספרים ממשיים, כל אחד לחוד וכולם יחד. בפרק 8, נראה את המילה cin המסמנת את ערוץ הקלט ומאפשרת קליטת נתונים מהמקלדת של המחשב.

סיכום

בפרק זה הצגנו מספר רכיבים המופיעים ברוב התוכניות בשפת C++. לפני שנעבור לפרק הבא, מומלץ לבחון אם מובנים הנושאים הבאים:

- ✓ כמעט כל התוכניות בשפת C++ מתחילות במשפט #include. משפט זה מורה למהדר להכניס במקומו את תוכן קובץ הכותר הרשום במשפט.
- ✓ קבצי כותר מכילים הגדרות שונות ושילובם בתוכניות מאפשר למתכנת לנצל אותן במהלך התוכנית.
- ✓ בקבצי מקור מספר רב של משפטים. המשפט void main(void) מציין את תחילת התוכנית הראשית, ולכן הפקודות שבתוכנית זו יתבצעו ראשונות בעת הרצת התוכנית.

✓ כאשר התוכנית נעשית ארוכה ומסובכת, ניתן לפשט ולשפר אותה על ידי חלוקתה לקטעי קוד קטנים, שכל אחד מהם בעל תפקיד מצומצם וברור. זוהי הגישה של **עבודה בפונקציות**. את משפטי הפונקציה כותבים בין סוגריים מסולסלים שמאליים וימניים: {<function statements...>}.

✓ ברוב התוכניות בשפת ++C נעזרים בערוץ הפלט cout להצגת הודעות ונתונים על המסך. על ידי שימוש בפקודות קלט/פלט של מערכת הפעלה ניתן לשנות את יעד הפלט של cout למקומות אחרים כגון קובץ, להתקנים אחרים כגון מדפסת, ואפילו לגרום לכך שנתוני פלט אלה ישמשו קלט לתוכנית אחרת.

כפי שלמדנו בפרק זה, ערוץ הפלט cout משמש את התוכנית להצגת הודעות ונתונים על המסך. בפרק 3 נראה כיצד ניתן בעזרתו להרכיב משפט פלט הכולל טקסט, מספרים שלמים ומספרים ממשיים. כמו כן, נראה את אופן העיצוב של פלטים אלה.

תרגילים

1. כיתבו תוכנית המנקה את המסך ומדפיסה בו את ההודעה הבאה:

```
This program clears the screen and prints this message!
```

כדי לנקות את המסך עליכם להורות למהדר לכלול בתוכנית את הספריה conio.h. עשו זאת באמצעות הפקודה include#. את המסך מנקים על ידי קריאה לפונקציית הספריה ;()clrscr, לפני הדפסת ההודעה.

2. נתבו למדפסת (במקום למסך) את פלט התוכנית שכתבתם עבור תרגיל 1 בפרק 1 (התוכנית המדפיסה את שמכם).

פרק 3

כתיבת הודעות על המסך

בפרקים הקודמים הראינו תוכניות המשתמשות בערוץ הפלט cout כדי להציג הודעות על המסך. בפרק זה נראה כיצד ניתן בעזרתו להרכיב משפט פלט הכולל מלל, מספרים שלמים (כגון 1010) ומספרים ממשיים (כגון 0.12345).

בהמשך הפרק נדון בנושאים הבאים:

- ערוץ הפלט cout משמש להצגת מספרים ותווים על המסך.
- ניתן להשתמש בתווים מיוחדים עם cout כדי לעבור לשורה חדשה, להשתמש בטבולטור אופקי, להפעיל את הרמקול הפנימי של המחשב ועוד.
- C++ מאפשרת להציג בקלות מספרים עשרוניים (Decimal), אוקטליים (בסיס 8 - Octal), או הקסדצימליים (בסיס 16 - Hexadecimal).
- על ידי שימוש באופרטורי הניתוב (Redirection operators) של מערכת ההפעלה ניתן לנתב את הודעות הפלט של תוכנית הנשלחות דרך cout, אל קובץ או אל מדפסת.
- ערוץ הפלט cerr מאפשר לתוכניות לשלוח הודעות לפלט הסטנדרטי, לדווח על תקלות, וכך למנוע מהמשתמשים לנתב מחדש את ההודעה.
- ניתן לעצב את פלט התוכנית על ידי שימוש בפונקציית העיצוב setw.

הצגת מספרים על ידי השימוש ב- cout

בתוכניות המוצגות בפרקים הקודמים השתמשנו ב-cout כדי להציג על המסך **מחרוזות תווים** (**Character strings**), שהן קבוצות של אותיות ומספרים הנמצאות בין גרשיים. בהמשך נראה שניתן להיעזר ב-cout להצגת מספרים. התוכנית הבאה, **1001.CPP**, מציגה את המספר 1001 על המסך.

```
#include <iostream.h>

void main(void)
 {
   cout << 1001;
 }
```

לאחר הידור והרצת התוכנית, יהיה הפלט המוצג על המסך 1001, כפי שנראה להלן:

```
C:\>1001 <Enter>
1001
```

כעת עליכם לערוך את התוכנית, ולשנות את משפט cout, כדי שעל המסך יופיע המספר 2002:

```
cout << 2002;
```

בנוסף להצגת מספרים שלמים (ללא ספרות עשרוניות), ניתן להיעזר ב- cout להצגת מספרים ממשיים כדוגמת 1.2345. מספרים אלה נקראים **מספרים בייצוג נקודת צפה** (**Floating-point numbers**). התוכנית הבאה, **FLOATING.CPP**, מציגה על המסך את המספר 0.12345:

```
#include <iostream.h>

void main(void)
 {
   cout << 0.12345;
 }
```

לאחר ההידור נריץ את התוכנית ונקבל על המסך את השורות הבאות:

```
C:\>FLOATING  <Enter>
0.12345
```

הצגת מספר ערכים בפקודה אחת

הסימן **קטן-מ** הכפול (>>) שמשתמשים בו בצירוף עם cout נקרא **אופרטור הקלט** או **אופרטור ההכנסה** (**Insertion operator**), מכיון שהוא **מכניס** את התווים והסימנים בתוך ערוץ הפלט. תחביר השפה מאפשרת שימוש במספר אופרטורים אלה בצירוף פקודת cout אחד. לדוגמה, התוכנית הבאה, **1001TOO.CPP**, נעזרת באופרטור הפלט ארבע פעמים כדי להציג את המספר 1001 על המסך.

```
#include <iostream.h>

void main(void)
 {
   cout << 1 << 0 << 0 << 1;
 }
```

לאחר הידור והרצת התוכנית, יוצגו על המסך השורות הבאות:

```
C:\>1001TOO  <Enter>
1001
```

בכל פעם ששפת ++C פוגשת באופרטור הקלט, מתבצע צירוף של המספרים או התווים הסמוכים לאופרטור בתוך ערוץ הפלט. התוכנית **SHOW1001.CPP**, מציגה מחרוזת ומספר על המסך, תוך שימוש ב- cout.

```
#include <iostream.h>

void main(void)
 {
   cout << "My favorite number is " << 1001;
 }
```

יש לשים לב, כי לאחר המילה is הקפדנו לשים תו-רווח, כדי לגרום לכך שבהצגת הפלט, המחרוזת לא תהיה **דבוקה** למספר (is1001). בצורה דומה, התוכנית **1001MID.CPP**, מציגה את המספר 1001 במרכז המחרוזת.

```
#include <iostream.h>

void main(void)
 {
   cout << "The number " << 1001 << " is my favorite";
 }
```

שימו לב לרווח שלפני ואחרי המספר 1001. כך נהגנו גם קודם.

לסיום, התוכנית הבאה, **MIXMATCH.CPP**, משלבת מחרוזות, תווים, מספרים שלמים ומספרי נקודה צפה בתוך אותו ערוץ הפלט:

```
#include <iostream.h>
void main(void)
 {
   cout << "At age " << 20 << " my salary was " << 493.34
                                                << endl;
 }
```

לאחר הידור והרצת תוכנית זו יופיע על המסך הפלט הבא:

```
C:\>MIXMATCH  <Enter>
At age 20 my salary was 493.34
```

השימוש בתווים מיוחדים

בכל התוכניות שכתבנו עד עתה רוכז הפלט בשורה אחת. אולם לפעמים, יש להציג הודעות פלט הפרוסות על פני מספר שורות. לדוגמה, היה רצוי כי תוכנית המציגה כתובת של אדם כלשהו, תתפרס על יותר משורה אחת. כעת נלמד כיצד ניתן להציג פלט המתפרס על מספר שורות.

נוכל ליצור שורה חדשה בשתי דרכים. בדרך אחת, מחדירים **תו שורה חדשה** (Newline character) - n\ במקום הרצוי במחרוזת (שים לב, התו \ מסמל שאחריו נמצא תו קוד, ובמקרה זה - n. התוכנית הבאה, **TWOLINES.CPP**, מדגימה הצגת פלט בשתי שורות, ושימוש בתו שורה חדשה:

```
#include <iostream.h>

void main(void)
 {
   cout << "This is line one\nThis is line two";
 }
```

לאחר הידור והרצת התוכנית יוצג הפלט על המסך בשתי שורות:

```
C:\>TWOLINES   <Enter>
This is line one
This is line two
```

במקרה שפקודת הפלט אינה כוללת מחרוזת, ניתן לצרף את התו **המיוחד שורה חדשה** בין אופרטורים הפלט, כאשר התו הוא בין גרשים בודדים ('n\'). לדוגמה, התוכנית **NEWLINES.CPP**, מציגה את הספרות 1, 0, 0 ו- 1, כל אחת בשורה נפרדת.

```
#include <iostream.h>

void main(void)
 {
   cout << 1 << '\n' << 0 << '\n' << 0 << '\n' << 1;
 }
```

ניתן להשתמש באופרטור **endl** (כמו גם בתו מיוחד **שורה-חדשה**) לקידום הסמן לשורה חדשה.

אופרטור endl מסמל **סוף שורה** (**end line**) ותחביר השימוש זהה ל**שורה-חדשה**. התוכנית **ENDL.CPP**, מתארת את השימוש ב**אופרטור endl** כדי לקדם את הסמן לתחילת שורה חדשה.

```
#include <iostream.h>

void main(void)
 {
   cout << "I've been..." << endl << "Rescued by C++";
 }
```

כמו קודם, לאחר הידור והרצת התוכנית יופיע הפלט על המסך בשתי שורות:

```
C:\>ENDL <Enter>
I've been...
Rescued by C++
```

לסיום, התוכנית הבאה, **ADDRESS.CPP**, מציגה במספר שורות את הכתובת של חברת Jamsa Press:

```
#include <iostream.h>

void main(void)
 {
    cout << "Jamsa Press" << endl;
    cout << "2975 South Rainbow, Suite I" << endl;
    cout << "Las Vegas, NV 89102" << endl;
 }
```

רשימת תווים מיוחדים

בנוסף לתו המיוחד **לשורה-חדשה**, בשפת ++C ניתן להשתמש בתווים מיוחדים אחרים המאפשרים ביצוע של מיגוון פעילויות. להלן טבלה 3.1 ובה רשימת התווים המיוחדים בשפה.

טבלה 3.1: תווים מיוחדים לשימוש בצירוף עם cout.

תו מיוחד	מטרה
\a	אזהרה או צפצוף (bell)
\b	backspace
\f	מעבר לדף חדש
\n	מעבר לשורה חדשה
\r	מעבר לתחילת השורה (ללא מעבר לשורה חדשה)
\t	טבולטור אופקי
\v	טבולטור אנכי
\\	הצגת הסימן \
\?	הצגת הסימן ?
\'	הצגת הסימן '
\"	הצגת הסימן "
\0	אפס - Null
\ooo	הצגת מספר בבסיס האוקטלי, לדוגמה: 007\
\xhhhh	הצגת מספר בבסיס ההקסדצימלי, לדוגמה: xFFFF\

הערה



כאשר משתמשים בתוך מחרוזת בתווים המיוחדים המופיעים בטבלה 3.1, יש לכתוב אותם במרכאות כפולות. כך לדוגמה: "Hello\n World!". כאשר כותבים את התווים המיוחדים לחוד (לא בתוך מחרוזת), יש לכתוב אותם בתוך מרכאות בודדות, כמו בדוגמה זו: '\n' .

התוכנית **SPECIAL.CPP** משתמשת בתווים מיוחדים \a ו- \t כדי להשמיע צפצופים ברמקול הפנימי של המחשב. בנוסף, היא מציגה את המחרוזת: Bell Bell Bell - על המסך, תוך שימוש בטבולטור אופקי.

```
#include <iostream.h>

void main(void)
 {
   cout << "Bell\a\tBell\a\tBell\a";
 }
```

הצגת מספרים בבסיס אוקטלי ובבסיס הקסדצימלי

בתוכניות שהראינו עד כה, הצגנו את המספרים בבסיס דצימלי. בעולם התכנות לפעמים יש צורך בהצגת המספרים בבסיסים שונים. הבסיסים הנפוצים הם בסיס אוקטלי (בסיס 8) ובסיס הקסדצימלי (בסיס 16). בתוכנית ++C ניתן להמיר מספרים ולהציגם על המסך בבסיסים שונים על ידי השימוש בפונקציות **המרת הפלט: dec**, **oct** ו-**hex**. לדוגמה בתוכנית הבאה, **OCTHEX.CPP**, מוצג השימוש בפונקציות המרת הפלט להצגת מספרים בבסיסים דצימלי, אוקטלי והקסדצימלי.

```
#include <iostream.h>

void main(void)
 {
  cout <<"Octal: " << oct << 10 << ' ' << 20 << endl;
  cout <<"Hexadecimal: " << hex << 10 << ' ' <<20<<endl;
  cout <<"Decimal: " << dec << 10 << ' ' << 20 << endl;
 }
```

לאחר ההידור והרצת התוכנית יוצגו על המסך השורות הבאות:

```
C:\>OCTHEX  <Enter>
Octal: 12 24
Hexadecimal: a 14
Decimal: 10 20
```

הערה



כאשר מנצלים את אחד ממעצבי הפלט להצגת פלט אוקטלי, הקסדצימלי או דצימלי, השימוש בו יהיה תקף עד סוף התוכנית, או עד השימוש במעצב אחר.

הפלט הסטנדרטי לדיווח תקלות cerr

כפי שציינו קודם, על ידי מתן פקודות מתאימות, מערכת ההפעלה מסוגלת לנתב את הפלט מהתקן הפלט הסטנדרטי אל התקנים אחרים. הדבר נכון כאשר מציבים בתוכניות בשפת ++C את הפקודה cout. ברם, כאשר מתרחשת תקלה בביצוע התוכנית, נרצה שדיווח התקלה יוצג על המסך ולא על התקן אחר, כדי להעיר את תשומת ליבו של המשתמש באותו רגע. לכן, במקרה של דיווח תקלות יש להציב בתוכנית את הפקודה cerr במקומה של cout, וכך להבטיח את השימוש בערוץ דיווח התקלות, שהוא ערוץ פלט המשויך תמיד למסך.

לדוגמה, התוכנית **CERR.CPP** מציגה על המסך את ההודעה:

```
This Message Always Appears
```

```
#include <iostream.h>

void main(void)
 {
   cerr << "This Message Always Appears";
 }
```

לאחר הידור, ניתן לנסות ולהריץ את התוכנית תוך שינוי התקן הפלט, כדלקמן:

```
C:\>CERR  >  FILENAME.EXT  <Enter>
```

במקרה זה, הודעת הפלט תופיע על המסך, על אף שינוי התקן הפלט. למעשה, הודעת הפלט מנווטת לערוץ הפלט הסטנדרטי לדיווח תקלות, שהוא המסך.

שים ♥! **ב-TCLite תוכנית זו לא תרוץ משורת הפקודה ב- DOS.**

קביעת אורך הפלט

בתוכניות הקודמות הצגנו מספרים על המסך והקפדנו לשים תווי רווח לפני ואחרי המספרים, כדי להבטיח הצגה נכונה וברורה של אותם המספרים. בשפת ++C קיימת פונקציית עיצוב בשם **setw** המאפשרת לקבוע את האורך המינימלי הדרוש להצגת פלט מסוים. משתמשים בפונקציה זו בצירוף פקודת cout, כאשר אורך הפלט נמדד ונקבע ב**תווים**.

לדוגמה, בתוכנית הבאה, **SETW.CPP**, מוצג השימוש בפונקציית setw להצגת המספר 1001 על המסך. כל פעם בה משתמשת התוכנית הבאה בפונקציה setw היא מוגדרת באורך אחר (3,4,5,6). כדי להשתמש בפונקציית setw יש להגדיר את קובץ הכותר iomanip.h בתחילת התוכנית.

```
#include <iostream.h>
#include <iomanip.h>

void main(void)
 {
  cout << "My favorite number is" <<setw(3)<<1001<<endl;
  cout << "My favorite number is" <<setw(4)<<1001<<endl;
  cout << "My favorite number is" <<setw(5)<<1001<<endl;
  cout << "My favorite number is" <<setw(6)<<1001<<endl;
 }
```

לאחר הידור והרצת התוכנית יוצגו על המסך השורות הבאות:

```
C:\>SETW  <Enter>
My favorite number is1001
My favorite number is1001
My favorite number is 1001
My favorite number is  1001
```

האורך שנקבע על ידי setw הוא המספר המינימלי של התווים הדרוש להצגת הפלט. בתוכנית הקודמת, המשפט (3) setw קובע אורך מינימלי של שלושה תווים להצגת המספר 1001, אולם אורך המספר הוא ארבעה תווים. במצבים אלה הפקודה cout תציב לפלט את האורך הדרוש (במקרה זה ארבעה תווים). יש לציין כי כאשר משתמשים בפונקציה setw, פעולתה תהיה תקפה לערך או למספר הסמוך לה. במקרה של משפט פלט בעל מספר רב של מופעים, יש להציב את הפונקציה לפני כל מופע ומופע במשפט.

הערה



התוכנית הקודמת השתמשה בקובץ הכותר iomanip.h, **מומלץ להדפיס את תוכן הקובץ ולקרוא אותו. גם קובץ כותר זה, כמו** iostream.h, **נמצא** בספריה INCLUDE.

סיכום

בפרק זה הצגנו כיצד פועלת הפקודה cout והראינו דרכים שונות לטיפול בפלט על המסך. בכל התוכניות המוצגות בספר זה השתמשנו בפקודה cout לטיפול בפלטים. לפני שנעבור לפרק הבא, מומלץ לבחון אם מובנים הנושאים הבאים:

- ✓ ערוץ הפלט cout מאפשר הצגה של מספרים, אותיות וסימנים.
- ✓ ניתן להשתמש בתו המיוחד **\n**, או במעצב הפלט endl כדי לקדם את הסמן לתחילת השורה הבאה.
- ✓ ניתן לשלוט על עיצוב ההדפסה (שורה חדשה, טבולטור אופקי וכו') על ידי הזנת תווים מיוחדים לערוץ הפלט.

- ✓ פונקציות המרה dec, oct ו- hex מאפשרות הצגת המספרים בבסיס דצימלי, אוקטלי והקסדצימלי בהתאמה. פונקציית העיצוב setw מאפשרת לקבוע את האורך המינימלי הרצוי להצגת ערכי הפלט.
- ✓ על ידי שימוש בערוץ הפלט cerr אפשר לדווח תקלות, ניתן להציג הודעות תקלה בהתקן המוגדר לכך (ברוב המקרים, המסך).
- ✓ ניתן לקבוע את מספר התווים המינימלי הדרוש להצגת ערך מסוים, על ידי שימוש במעצב הפלט setw.

תרגילים

1. כיתבו תוכנית אשר הפלט שלה הוא:

```
1002
0.707
-419
```

2. כיתבו תוכנית אשר תדפיס את שורות הפלט הבאות:

```
3 * 7 = 21
I like to eat 4 chocolates every day
```

התוכנית צריכה להשמיע שני צפצופים (beep) בסיום ההדפסה.

3. כיתבו תוכנית המדפיסה את ערך המספר 19 בבסיס הקסדצימלי ובבסיס אוקטלי. התוכנית צריכה להדפיס שורה מהצורה:

```
19 is 0x13 hexadecimal and 023 octal
```

יש להשתמש באופרטור המרת פלט להדפסת המספרים.

4. כיתבו תוכנית המדפיסה לערוץ דיווח התקלות את הודעת השגיאה הזו:

```
This is an Error message!!!
```

5. כיתבו תוכנית המדפיסה (דרך cout) את הפלט הבא, ברווחים 3,5,7,9, בעזרת הפונקציה setw:

```
123
123
123
123
```

פרק 4

משתנים לאחסון המידע

עד כאן הצגנו תוכניות פשוטות מאוד. כאשר כותבים תוכנית המבצעת משימות בעלות משמעות, מתעורר הצורך בשמירת מידע במהלך הרצת התוכנית. למשל, תוכנית המדפיסה תוכנו של קובץ מסוים, ודאי צריכה לדעת את שמו של הקובץ ובמקרים מסוימים את מספר העותקים הרצוי להדפסה. במהלך הרצת התוכנית, המידע הדרוש מאוחסן בזיכרון הפנימי, או זיכרון העבודה, של המחשב. התוכניות משתמשות ב**משתנים** (**Variables**) כדי לגשת ולהתייחס לנתונים מסוימים המאוחסנים בזיכרון. במילים פשוטות ניתן להגדיר, כי משתנה הוא שם של מקום בזיכרון בו ניתן לאחסן ערכים מסוימים.

בהמשך הפרק נדון בנושאים הבאים:

- צריך לציין למהדר מהם המשתנים בהם ייעשה שימוש במהלך התוכנית על ידי הצהרה על השם והטיפוס (סוג, type).
- טיפוס של משתנה קובע את אוסף הערכים שהמשתנה יכול להכיל (למשל ערכים שלמים,או ערכים מטיפוס נקודה צפה), וגם את הפעולות שניתן לבצע עליו.
- אופרטור ההשמה של ++C משמש להקצאת ערכים למשתנים.
- יש להשתמש בערוץ הפלט cout כדי להציג על המסך את ערך המשתנה.
- קביעת שמות בעלי משמעות למשתנים, תקל עליך ועל מתכנתים אחרים לקרוא ולהבין את התוכנית.
- תיעוד התוכנית לכל אורכה בעזרת הערות ברורות המתארות פעולות שונות, תקל עליך ועל מתכנתים אחרים, כאשר צריך לעדכן אותה בעתיד.

באופן ציורי ניתן לדמות משתנה לתיבה, שבה ניתן לשים ערכים. כאשר התוכנית זקוקה לערך המשתנה, היא פשוט ניגשת לתיבה ומתבוננת בערך שבתוכה.

הצהרה על משתנים בתוכנית

לא כל משתנה מסוגל לאחסן בתוכו כל ערך. התוכניות מסוגלות לטפל בסוגים שונים של ערכים: מספרים שלמים, אותיות האלף-בית, מספרים ממשיים. בתוכניות C++ מאחסנים את ערכים במשתנים המתאימים לסוגם, לכן ניתן לחלק את המשתנים לפי טיפוסים שונים. להלן טבלה 4.1 המציגה את טיפוסי המשתנים הנפוצים בשפת C++.

טבלה 4.1: רשימת טיפוסי משתנים נפוצים בשפת C++

טיפוס	**ערך לאחסון**
char	ערכים בתחום: 128- עד 127. בדרך כלל משמש לאחסון אותיות.
int	ערכים בתחום: 32,768- עד 32,767.
unsigned	ערכים בתחום: 0 עד 65,535.
long	ערכים בתחום: 2,147,483,648- עד 2,147,483,647
float	ערכים בתחום: $-3.4 * 10^{38}$ עד $3.4 * 10^{38}$
double	ערכים בתחום: $-1.7 * 10^{308}$ עד $1.7 * 10^{308}$

כדי להשתמש במשתנה, מחייב תחביר השפה ל**הצהיר** עליו (**Declare**), או במילים אחרות – להגדיר אותו בתוכנית. הצהרת המשתנה כוללת טיפוס המשתנה והשם שבו הוא יופיע בתוכנית. בדרך כלל, הצהרת המשתנים נרשמת בתחילת התוכנית אחרי הסוגריים המסולסלים. להלן תבנית התחביר להצהרת משתנה:

```
variable_type  variable_name;
```

רוב המשתנים הם מהטיפוסים הרשומים בטבלה 4.1. מכיון שהמתכנת קובע את שם המשתנה, מומלץ ורצוי שיהיה זה שם בעל משמעות אשר יתאר את תפקיד המשתנה, כמו למשל, employee_name (שם-עובד) או employee_age (גיל-עובד). C++ מתייחסת אל הצהרות משתנים כאל משפטים, ולכן צריך לכתוב נקודה-פסיק (**;**) בסוף ההצהרה.

להלן דוגמה להגדרת מספר משתנים. נשים לב שמשתמשים בנקודה-פסיק להפרדה של הצהרת משתנים מטיפוסים שונים.

```
#include <iostream.h>
void main(void)
 {
   int test_score;
   float salary;
   long distance_to_mars;
 }
```

התוכנית שבדוגמה אינה מבצעת כל פעולה, זולת הצהרת המשתנים. ניתן להבחין במיקום סימני נקודת-פסיק, המפרידים בין הצהרות טיפוסים שונים של משתנים.

להצהרת מספר משתנים בעלי אותו טיפוס יש לרשום פסיק בין שמות המשתנים הרצויים. להלן דוגמה להצהרת שלושה משתנים מטיפוס ממשי.

```
float salary, income_tax, retirement_fund;
```

מה זה משתנה?

משתנה הוא שם של מקום בזיכרון העבודה של המחשב בו מאוחסנים נתונים. התוכנית במהלך הביצוע מאחסנת במשתנים את המידע הדרוש. בשפת ++C חייבים להצהיר על משתנים שבשימוש התוכנית. ההצהרה מתבצעת על ידי ציון טיפוס המשתנה ושמו. להלן דוגמה להצהרת משתנה age מטיפוס שלם (int):

```
int age;
```

קביעת שמות בעלי משמעות למשתנים

המשתנים המוגדרים בתוכנית חייבים להיות בעלי שמות ייחודיים. כדי שהתוכנית תהיה יעילה וברורה, מומלץ לתת למשתנים שמות בעלי משמעות המתארים את התפקיד שלהם. למשל, בתוכנית מסוימת מוגדרים המשתנים הבאים:

```
int x, y, z;
```

בהנחה שהמשתנים לעיל מתייחסים לגיל התלמיד, ציון ממוצע וציון בבחינה, עדיף יהיה להצהיר עליהם כך:

```
int student_age, test_score, grade;
```

שמות המשתנים יכולים להכיל צירופי אותיות, מספרים וסימן קו-תחתון (_). הסימן הראשון בשם המשתנה חייב להיות אות או קו-תחתון, ובכל מקרה אסור ששם משתנה יתחיל במספר. שפת ++C מבדילה בין אותיות גדולות ובין אותיות קטנות. לכן, בשלבים הראשונים של לימוד השפה, מומלץ להגדיר את שמות המשתנים תמיד באותיות קטנות. בהמשך ניתן להשתמש באותיות גדולות לשיפור הבהירות של התוכניות. לדוגמה:

```
float MonthlySalary, IncomeTax;
```

מילים שמורות

בנוסף לכללי מתן השמות שהוצגו בפיסקה הקודמת, קיימות בשפת C++ מספר מילים שאסור להשתמש בהם כשמות משתנים. מילים אלו הן מילים שמורות. רשימת המילים השמורות בשפת C++ ניתנת בטבלה 4.2.

טבלה 4.2: רשימת מילים שמורות בשפת C++.

asm	auto	break	case	catch	char
class	const	continue	default	delete	do
double	else	enum	extern	float	for
friend	goto	if	inline	int	long
new	operator	private	protected	public	register
return	short	signed	sizeof	static	struct
switch	template	this	throw	try	typedef
union	unsigned	virtual	void	volatile	while

מדוע דרושים משתנים בתוכנית?

ככל שתוכניות מורכבות יותר, כן גדל מספר הפעולות שמתבצעות בהן על פריטים שונים. למשל, תוכנית המחשבת משכורות עובדים, צריכה לעבד את נתוניו של כל עובד. נניח, כי התוכנית משתמשת במשתנים (Variables) שם-עובד, מספר-ת.ז. ומשכורת (כמובן שתכתוב את המשתנים באנגלית, כנדרש בתכנות). כשהתוכנית תופעל, היא תקצה את נתוני העובד הראשון למשתנים הללו ותפעל בעזרתם לחישוב משכורתו. לאחר מכן, היא תחזור על תהליך זה עבור כל אחד מהעובדים האחרים, ותציב עבור כל אחד מהם את פרטיו האישיים במשתנים הללו. במילים אחרות, בעת הריצה, התוכנית מקצה ערכים חדשים ושונים למשתנים הללו, ומשתמשת בהם בחישוב משכורתו של העובד הנוכחי. המשתנים האלה (כולם, או חלקם) מכילים ערכים שונים בכל שלב חישוב, ולכן הם נקראים בשם זה.

הקצאת ערך למשתנה

במהלך התוכנית ניתן לאחסן או להקצות ערכים למשתנים. כדי לעשות זאת, משתמשים ב**אופרטור ההשמה** (**Assignment operator**), סימן שווה. בשורות הבאות להלן מוצגות מספר דוגמאות להקצאת ערכים למשתנים מטיפוסים שונים. יש לשים לב שבסיום כל משפט השמה מופיע הסימן נקודה-פסיק.

```
age = 32;
salary = 25000.75;
distance_to_the_moon = 238857L;
```

הערה 

הערכים הניתנים למשתנים אינן כוללים סימני פיסוק של מספרים (פסיקים או נקודות) כדוגמת 25,000.75 או 238,857. רישום הסימנים האלה יגרום להפסקת ההידור של התוכנית והצגת הודעת שגיאה מתאימה.

בתוכנית הבאה יש דוגמאות להצהרת משתנים והשמת ערכים בהם:

```
#include <iostream.h>

void main(void)
 {
    int age;
    float salary;
    long distance_to_the_moon;

    age = 32;
    salary = 25000.75;
    distance_to_the_moon = 238857L;
 }
```

הקצאת ערכים בעת הצהרת המשתנים

לעיתים קרובות צריך לקבוע ערך התחלתי למשתנים. שפת C++ מאפשרת להקצות ערכים למשתנים בעת ההצהרה עליהם. להלן דוגמה:

```
int age = 32;
float salary = 25000.75;
long distance_to_the_moon = 238857L;
```

ברוב התוכניות הנכללות בספר ניתנים ערכים התחלתיים למשתנים במסגרת הצהרתם.

הקצאת ערך למשתנה

במהלך התוכנית המשתנים מאחסנים בתוכם ערכים. בשפת C++ משתמשים באופרטור ההשמה (סימן שווה) להקצאת ערך למשתנה. במשפט הבא מתבצע השמה של הערך 4 למשתנה בשם lesson, תוך שימוש באופרטור ההשמה:

```
lesson = 4;
```

בשפת C++ ניתן לקצר את הדרך להקצאת ערכים התחלתיים למשתנים, על ידי שימוש באופרטור ההשמה במסגרת משפט הצהרת המשתנה, כדלקמן:

```
int lesson = 4;
```

השימוש בערכי המשתנים

לאחר שמשתנה מקבל ערך ניתן להתייחס לערך זה (לעבד ולקרוא אותו) על ידי שימוש בשם המשתנה המכיל אותו. לדוגמה, בתוכנית הבאה **SHOWVARS.CPP**, מתבצעת הקצאת ערכים לשלושת המשתנים שהזכרנו ולאחר מכן ערכיהם מוצגים על המסך כהודעות פלט, תוך שימוש בפקודה cout.

```
#include <iostream.h>

void main(void)
 {
  int age = 32;
  float salary = 25000.75;
  long distance_to_the_moon = 238857L;
  cout << "The employee is " << age <<"years old" <<endl;
  cout << "The employee makes $" << salary << endl;
  cout << "The moon is " << distance_to_the_moon <<
                        " miles from the earth" << endl;
 }
```

הערה



משפט ה- cout האחרון בתוכנית אינו נכלל בשורה אחת, ועל כן **פיצלנו** אותו לשתי שורות. בשפת ++C ניתן "לפצל" משפטים בצורה זו, מכיון שסיום המשפט מסומן על ידי התו נקודה-פסיק. בכל מקרה, מומלץ לפצל משפטים בנקודת השבירה שלהם (כמו תחילת מחרוזת, סוגריים או אופרטור) ולהזיח את השורה החדשה לימין בשניים או שלושה תווים כמו בדוגמה.

כאשר מהדרים ומריצים תוכנית זו, המסך מציג את הפלט הבא:

```
C:\>SHOWVARS <Enter>
The employee is 32 years old
The employee makes $25000.75
The moon is 238857 miles from the earth
```

כפי שניתן לראות, כדי **לנצל** את הערך שמאוחסן במשתנה, צריך לציין את שם המשתנה במשפטי התוכנית.

גלישת ערכי משתנים

כפי שלמדנו, הטיפוס (סוג) של משתנה קובע איזה ערכים הוא יכול לאחסן. לדוגמה, משתנה מטיפוס int יכול להכיל ערכים בתחום מ- 32,768- ועד 32,767. הקצאת ערך שאינו בטווח הערכים האלה למשתנה int תגרום **שגיאת גלישה** (Overflow error). התוכנית הבאה, **OVERFLOW.CPP**, ממחישה מהי שגיאת גלישה. היא מקצה למשתנים ערכים שאינם בטווח המתאים להם. בתוכנית יש שני Warnings מכוונים.

```
#include <iostream.h>

void main(void)
 {
  int positive = 40000;
  long big_positive = 4000000000L;
  char little_positive = 210;

  cout << "positive now contains " << positive << endl;
  cout << "big_positive now contains " << big_positive << endl;
  cout <<"little_positive now contains "
                             << little_positive << endl;
 }
```

לאחר הידור והרצת התוכנית נראה את הפלט הבא על המסך:

```
positive now contains -25536
big_positive now contains -294967296
little_positive now contains ò
```

הערה



ערך התו little_positive יכול להשתנות בהתאם לסביבת ההפעלה, DOS או Windows.

כפי שניתן לראות, התוכנית הקצתה למשתנים ערכים מסוג long, int ומסוג char, שאינם בטווח המתאים להם, ולכן נוצרה שגיאת גלישה. שגיאות גלישה קשות לאיתור, ולכן יש לזכור תמיד בזמן התכנות מהם תחומי הערכים המתאימים לכל משתנה.

שים ♥ !

הערך שהתוכנית מציגה עבור המשתנה little_positive הוא מסוג char (תו), מכיון שהמשתנה הוגדר מטיפוס זה. ומכיון שכך, התו שהוצג על ידי cout כערכו של המשתנה little_positive מתאים לתו ASCII המורחב שערכו 210.

דיוק של ערכי משתנים

כפי שלמדנו, כאשר מקצים למשתנים ערכים אשר חורגים מטווח הערכים שהם יכולים להכיל, קורות שגיאת גלישה. באופן דומה, צריך לתת את הדעת גם לבעיית יכולת המחשב להציג ערכים מדויקים של מספרים בעלי ספרות עשרוניות. זוהי בעיית **דיוק** (Precision) בעת אחסון (וכתוצאה מכך – הצגה) של ערכים מספריים. כאשר עובדים, למשל, במספרים מטיפוס נקודה צפה (Floating point - מספרים בעלי נקודה עשרונית), המחשב אינו יכול להציג אותם תמיד בדיוק של מספר הספרות העשרוניות שלהם. **שגיאות דיוק** (Precision errors) כאלו קשות מאוד לגילוי.

התוכנית הבאה, **PRECISE.CPP**, מקצה ערך קטן מ- 0.5 למשתנה מטיפוס float, double (כלומר: נקודה צפה, דיוק כפול). התוכנית ממחישה את מגבלות המחשב בעת ייצוג מספרים. המשתנה בתוכנית אינו מכיל את הערך שהוצב בו, אלא את הערך 0.5:

```
#include <iostream.h>

void main(void)
 {
  float f_not_half = 0.49999990;
  double d_not_half = 0.49999990;

  cout << "Floating point 0.49999990 is " << f_not_half
                                                << endl;
  cout << "Double 0.49999990 is " << d_not_half << endl;

 }
```

לאחר הידור והרצת התוכנית יופיע על המסך הפלט הבא:

```
Floating point 0.49999990 is 0.5
Double 0.49999990 is 0.5
```

כפי שניתן לראות, הערכים שהוקצו למשתנים והערכים שהמשתנים מכילים למעשה **אינם זהים**. אי דיוקים אלה קורים מפני שקוד המכונה של המחשב מייצג מספרים באמצעות מספר קבוע של ספרות אחד ואפס (מילה בגודל קבוע). אין להבין מהכתוב, שהמחשב אינו יכול כלל להציג ערכים באופן מדויק, אלא, שבעת צורך להשתמש במספרים בעלי רמת דיוק גבוהה, יש לזכור את בעיית אי-הדיוק האפשרית ולטפל בה בכלים מתאימים, מכיון שקשה מאוד לאתר שגיאות מסוג זה. בנושא זה לא נעסוק במסגרת הספר.

שימוש בהערות
לשיפור קריאות התוכנית

בתוכניות מורכבות המכילות מספר רב של פקודות לא תמיד כל המשפטים ברורים, או קלים להבנה. לפעמים יש גם צורך להעביר למתכנתים אחרים את התוכניות שנוצרו על ידי עמיתיהם בצוותי התכנות של פרויקט מסוים. על כן חשוב להקפיד על כתיבה ברורה של התוכניות. הנה מספר דרכים מומלצות לשיפור קריאות התוכנית:

- ❖ שימוש בשמות משתנים בעלי משמעות המתארים את תפקיד המשתנה.
- ❖ עריכה והזחה (Indentation) נכונה של המשפטים בתוכנית (ראה פרק 7).
- ❖ שימוש בשורות ריקות להפרדה בין קבוצות של משפט ביצוע.
- ❖ שימוש בהערות בתוך התוכנית להסבר תהליכי התוכנית.

ניתן לציין שתי מעלות לשימוש ב**הערות** (**Comments**) בתוך התוכנית. הראשונה – הן מאוד מועילות למתכנתים, כאשר הם מעיינים בתוכניות שנכתבו על ידי מתכנת אחר. השנייה – כאשר הזמן חולף, וכותב התוכנית חייב להיזכר בפרטי הדברים שהוא עצמו צירף לתוכנית. להכנסת הערות בתוך התוכנית בשפת ++C משתמשים בתווים // (שני לוכסנים), כמו בדוגמה:

```
// This is a comment
```

שני הלוכסנים // מציינים למהדר שלא להתייחס למשפטים הכתובים אחריהם ועד לסוף השורה. אין זה מחייב, אך רצוי בתחילת כל קובץ מקור לכתוב מספר מילים אודות שם התוכנית, שם המתכנת, תאריך הביצוע וכמובן מטרת התוכנית:

```
// Program: BUDGET.CPP
// Programmer: Kris Jamsa
// Date Written: 1-10-99
//
// Purpose: Tracks monthly budget information.
```

בהמשך התוכנית ניתן לבאר את המשפטים השונים, תוך מיקום הערות הסבר אחרי המשפטים או תחתיהם.

למשל, נתייחס למשפט הבא:

```
  distance_to_the_moon = 238857L;    // Distance in miles
```

הערה שהוצבה מימין למשפט מתארת בבירור את המידע הנוסף אודות הביצוע שלו. סוגיה מעניינת היא, מה הוא הגבול להצבת הערות? ניתן להשיב בפשטות: אף פעם אין די הערות בתוכנית, אולם אין להיגרר למצבים מוגזמים הגובלים בגיחוך. ההערות המוצגות בשורות הבאות אינן מוסיפות כלל למעיין בתוכנית.

```
age = 32;    // Assign 32 to the variable age
salary = 2500.75; // Assign 2500.75 to the variable salary
```

מטרת הערות בגוף התוכנית היא להסביר מדוע מוצבים המשפטים במקומם.

תיעוד פנימי של התוכנית

ברוב התוכניות קיים צורך בהסברים פנימיים ותיעוד התהליכים המתבצעים בהן. כך יכולים המתכנת עצמו וגם מתכנתים אחרים להבין טוב יותר את התוכניות ולבצע בהן שינויים בעת הצורך. בשפת ++C משתמשים בשני לוכסנים (//) לציון הערה פנימית בתוכנית. הנה דוגמה:

```
// This is a c++ comment
```

בתהליך ההידור, כאשר המהדר מזהה את הסימנים //, הוא מתעלם מהמילים הכתובות אחריהם מימין עד סוף השורה. תוכניות הכתובות היטב מצטיינות בקלות הבנתן. הערות בתוך התוכנית מגבירות את הבהירות של תוכנית.

הערה



בנוסף לשימוש **בהערות** (Comments) כדי לשפר את קריאות התוכנית, צריך להשתמש בשורות ריקות כדי להפריד בין קטעי תוכנית או בין משפטי תוכנית שאינם "שייכים" זה לזה. כאשר מהדר ++C מגלה שורה ריקה, הוא פשוט מדלג עליה.

סיכום

בפרק זה הצגנו כיצד ניתן לאחסן מידע בתוך משתנים במהלך ביצוע התוכנית. בקצרה, **משתנה** (Variable) הוא שם שהתוכנית מקצה למקום הזיכרון שבו היא שומרת נתונים או תוצאות. לפני שהתוכנית משתמשת במשתנה, צריך להגדיר שם וסוג שניתנים לו. בפרק 5 תלמד כיצד לבצע במשתנים פעולות חישוב פשוטות, כמו חיבור וחיסור.

לפני שנעבור לפרק הבא מומלץ לבחון אם מובנים הנושאים הבאים:

- ✓ כדי להשתמש במשתנים צריך להצהיר את טיפוס המשתנה ואת שמו.
- ✓ שמות המשתנים חייבים להיות ייחודיים, ורצוי שיהיו בעלי משמעות.
- ✓ שמות המשתנים מתחילים באותיות, או בקו-תחתון.
- ✓ שפת ++C מבדילה בין אותיות גדולות לבין אותיות קטנות.
- ✓ טיפוס המשתנה קובע איזה ערכים ניתן לאחסן במשתנה. טיפוסי משתנים נפוצים בשפת ++C הם: float, int, char ו- long.
- ✓ הערות בתוך התוכנית משפרות את הבהירות של התוכנית. בשפת ++C שני תווי לוכסן (//) מציינים הערה בתוך התוכנית.

תרגילים

1. כיתבו תוכנית המגדירה את המשתנים הבאים:

   א) מידת נעליים - משתנה מסוג int

   ב) האות הראשונה בשם - משתנה מסוג char

   ג) מספר הכוכבים בדגל ארה״ב - משתנה מסוג int

   ד) מספר האטומים ביקום - משתנה מסוג double

   ה) ערך הקבוע π - משתנה מסוג float

   לאחר מכן יש לקבוע ערכים התחלתיים כלשהם לכל המשתנים (פרט למשתנה האחרון, המקבל את הערך 3.14), ולהדפיס אותם.

2. כיתבו תוכנית המחשבת נפח קוביה. הגדירו משתנה מסוג int, אשר ערכו הוא צלע הקוביה, ואתחלו אותו לערך 10. יש להשתמש במשתנה נוסף מסוג int לחישוב נפח הקוביה, ולהדפיס את ערכו. כעת תנו ערך חדש למשתנה אורך הצלע, חשבו את הנפח החדש, והדפיסו את התוצאה שמתקבלת.

3. מיצאו את השגיאות בתוכנית הבאה:

```
#include <iostream.h>

void main(void)
  {
       iNt  x;
       long y 459;
       float f = 32,459.71
       double d1 = 2  d2 = 3;
       char ;
       int 43aa;
       long public;
       float r, r;
  }
```

4. התוכנית הבאה בודקת תוצאות השמה בין סוגי משתנים שונים. הגדירו משתנה בשם f מסוג float בעל ערך 4.7. כעת הגדירו משתנה נוסף בשם i מסוג int שלתוכו יש להציב את ערך המשתנה f. הדפיסו את הערך המתקבל במשתנה i.
   במקביל לסעיף הקודם, יש לתת למשתנה חדש j את הערך 5, ולהציב את ערך המשתנה j לתוך משתנה f. יש להדפיס את הערך המתקבל ב-f. התוכנית צריכה לכלול תיעוד, כלומר - הסברים/הערות.

5. התרגיל הבא בודק את מגבלות הדיוק של המחשב. הגדירו משתנה מסוג int ואתחלו אותו לערך 12345678. הגדירו משתנה מסוג float ואתחלו אותו לערך 7 כפול 10 בחזקת 41- (7.0E-41). יש להדפיס את הערכים המתקבלים.

   הערה: *בתרגיל יש שני Warnings וזה בסדר.*

פרק 5

פעולות אריתמטיות בסיסיות

בפרק הקודם הראינו כיצד מצהירים על משתנים ומגדירים אותם בתוכנית. בדרך כלל, בתוכניות מחשב משתמשים בערכים המאוחסנים במשתנים כדי לבצע פעולות אריתמטיות כגון סיכום, חיסור, כפל וחילוק.

בהמשך הפרק נדון בנושאים הבאים:

- שימוש באופרטורים אריתמטיים של C++ לביצוע פעולות חשבוניות בתוכנית.
- C++ קובעת סדר קדימויות לאופרטורים האריתמטיים השונים, כדי להבטיח שפעולות חשבוניות תבוצענה בסדר נכון.
- שליטה על סדר הפעולות החשבוניות בתוכנית באמצעות סוגריים.
- ניתן להשתמש באופרטור ההגדלה העצמית (++) כדי לחבר 1 לערכו של משתנה, ובאופרטור ההקטנה העצמית (--) כדי לחסר 1 מערכו של משתנה.

לאחר הצגת נקודות אלו, נוכל לראות כמה קל לבצע פעולות אריתמטיות בשפת C++!

אופרטורים מתמטיים בסיסיים

ברוב התוכניות צריך לערוך חישובים אריתמטיים בערכים ובמשתנים. הפעולות האריתמטיות הבסיסיות הן חיבור, חיסור, כפל וחילוק. בהמשך נציג כיצד לבצע פעולות אלו על ערכים קבועים (דוגמה: 3 * 5) ועל משתנים (כמו: total + sum).

בטבלה 5.1 מוצגת רשימת האופרטורים האריתמטיים הבסיסיים.

טבלה 5.1: אופרטורים אריתמטיים בסיסיים בשפת ++C.

האופרטור	פעולה	דוגמה
+	חיבור	total = cost + tax;
-	חיסור	change = payment - total;
*	כפל	tax = cost * tax_rate;
/	חילוק	average = total / count;

התוכנית **SHOWMATH.CPP** משתמשת בפקודה cout להצגת התוצאות של מספר חישובים אריתמטיים בסיסיים.

```
#include <iostream.h>

void main(void)
 {
   cout << "5 + 7 = " << 5 + 7 << endl;
   cout << "12 - 7 = " << 12 - 7 << endl;
   cout << "1.2345 * 2 = " << 1.2345 * 2 << endl;
   cout << "15 / 3 = " << 15 / 3 << endl;
 }
```

כשנתבונן במשפטי התוכנית, נוכל להבחין שבכל שורה מופיע הביטוי האריתמטי פעמיים: פעם אחת בין גרשיים, כדי שהתוכנית תציג את הביטוי ופעם שנייה ללא גרשיים, כדי להציג את התוצאה עצמה. בהמשך המשפט מופיע סימן של שורה חדשה. לאחר הידור והרצת התוכנית יופיעו על המסך השורות הבאות:

```
C:\>SHOWMATH  <Enter>
5 + 7 = 12
12 - 7 = 5
1.2345 * 2 = 2.469
15 / 3 = 5
19 / 4 = 4
```

במקרה הקודם הצגנו פעולות מתמטיות כאשר האופרנדים הם ערכים קבועים. בתוכנית **MATHVARS.CPP** מוצגות פעולות אריתמטיות אשר מופעלות על משתנים.

```
#include <iostream.h>

void main(void)
 {
  float cost = 15.00;         // The cost of an item
  float sales_tax = 0.06;     // Sales tax is 6 percent
  float amount_paid = 20.00;  // The amount the buyer paid
  float tax, change, total;   // Sales tax,buyer change and
                              // total bill

  tax = cost * sales_tax;
  total = cost + tax;
  change = amount_paid - total;

  cout << "Item Cost: $" << cost << "\tTax: $" << tax
                       << "\tTotal: $" << total << endl;

  cout << "Customer change: $" << change << endl;
 }
```

המשתנים המופיעים בתוכנית הם מטיפוס ממשי. ניתן לראות, כי בהצהרה על המשתנים מתבצעת השמה של ערכים רלוונטיים. בהמשך, התוכנית מבצעת סדרה של פעולות אריתמטיות לחישוב מס הקנייה של המוצר, העלות הכוללת של המוצר והעודף המגיע לקונה. לאחר הידור והרצת התוכנית יוצג על המסך הפלט הבא:

```
C:\>MATHVARS <Enter>
Item cost: $15.0            Tax: $0.93     Total: $15.9
Customer change: $4.1
```

קידום ערך המשתנה באחד

אחת הפעולות הנפוצות בתוכניות מחשב היא קידום ערכי משתנים באחד. בדרך כלל משתנים אלה משמשים **מונים** לביצוע פעולות כדוגמת ציון מספר הקובץ להדפסה, או ספירה של אירועים. להלן דוגמה למשפט בשפת ++C המקדם את ערך המשתנה count באחד, כלומר, לאחר ביצוע המשפט ההשמה. הערך המאוחסן במשתנה count יהיה תוצאה של חיבור הערך הקודם ועוד אחד.

```
count = count + 1;
```

התוכנית **INCCOUNT.CPP** משתמשת באופרטור ההשמה לקידום ערך המשתנה num באחד.

```
#include <iostream.h>

void main(void)
 {
   int num = 1000;
   cout << "count's starting value is " << num <<endl;
   num = num + 1;
   cout << "count's ending value is " << num << endl;
 }
```

לאחר הידור והרצת התוכנית, יוצגו על המסך השורות הבאות :

```
C:\>INCCOUNT  <Enter>
count's starting value is 1000
count's ending value is 1001
```

מכיון שפעולת קידום ערך המשתנה באחד שכיחה מאוד, קיים בשפת ++C אופרטור מיוחד לפעולה זו ושמו **אופרטור להגדלה עצמית** (**Increment operator**) האופרטור מורכב משני סימני פלוס (++) הצמודים למשתנה. להלן שני משפטים, המבצעים הגדלת ערך המשתנה num באחד :

```
num = num + 1;                    num++;
```

התוכנית הבאה **INC_OP.CPP** משתמשת באופרטור להגדלה עצמית, כדי לקדם את ערך המשתנה num באחד :

```
#include <iostream.h>
void main(void)
 {
   int num = 1000;
   cout << "count's starting value is " << num <<endl;
   num++;
   cout << "count's ending value is " << num << endl;
 }
```

תוכנית זו פועלת בצורה זהה לתוכנית INCCOUNT.CPP שראינו קודם.

משמעות מיקום האופרטור להגדלה עצמית ביחס למשתנה

ניתן להציב את האופרטור להגדלה עצמית לפני שם משתנה המיועד לקידום, או אחריו. לדוגמה :

```
++variable;                       variable++;
```

במקרה הראשון, כשהאופרטור מופיע **לפני** שם המשתנה הוא נקרא **אופרטור הגדלה עצמית מוקדמת** (**Prefix increment operator**). במקרה השני, כשהוא מופיע **אחרי** שם המשתנה הוא נקרא **אופרטור הגדלה עצמית מאוחרת** (**Postfix increment operator**). הבה נבין את ההבדל בפעולות האופרטור לפי המיקום שלו ביחס לשם המשתנה. לפנינו הביטוי הבא:

```
current_count = count++;
```

במשפט ההשמה מתבצעת הקצאת הערך הנוכחי של משתנה count בתוך המשתנה current_count. בנוסף ולאחר ביצוע ההשמה, אופרטור הגדלה עצמית "מאוחר" גורם לקידום ערך המשתנה count באחד. לכן, המשפט הקודם שקול לשני המשפטים הבאים:

```
current_count = count;
count = count + 1;
```

כעת נעבור לאופרטור קדם הגדלה עצמית. נתייחס לביטוי הבא:

```
current_count = ++count;
```

במקרה זה, לפני שמבצעים את משפט ההשמה, מגדילים את ערך המשתנה count באחד. את התוצאה מקצים למשתנה current_count. לכן המשפט הקודם שקול לשני המשפטים שלהלן.

```
count = count + 1;
current_count = count;
```

כאמור, קידום משתנים באחד הינה פעולה נפוצה מאוד בתוכניות C++, ולכן חשוב מאוד לדעת את אופן הפעולה של האופרטור להגדלה עצמית בשתי הגרסאות שלו. בתוכנית הבאה **PRE_POST.CPP**, מוצג השימוש באופרטור להגדלה עצמית:

```
#include <iostream.h>

void main(void)
 {
  int small_count = 0;
  int big_count = 1000;

  cout << "small_count is " << small_count << endl;
  cout << "small_count++ yields " << small_count++ <<
endl;
  cout << "small_count ending value"<<small_count << endl;

  cout << "big_count is " << big_count << endl;
  cout << "++big_count yields " << ++big_count << endl;
  cout << "big_count ending value " << big_count << endl;
 }
```

לאחר הידור והרצת התוכנית יוצגו על המסך השורות הבאות :

```
C:\>PRE_POST  <Enter>
small_count is 0
small_count++ yields 0
small_count ending value 1
big_count is 1000
++big_count yields 1001
big_count ending value 1001
```

בתוכנית הקודמת מופעל אופרטור הגדלה עצמית מאוחרת על המשתנה small_count. כתוצאה מפעולה זו התוכנית מציגה את הערך המקורי של המשתנה, שהוא אפס, ואת הערך החדש שהוא אחד. לעומת זאת, על המשתנה big_count מופעל האופרטור קדם הגדלה עצמית. התוכנית מקדמת את ערכו באחד טרם ביצוע משפט ההצגה, ולכן מוצג על מסך הערך 1001 שהוא ערכו החדש של המשתנה.

אופרטור הקטנה עצמית

ראינו קודם, כי בשפת ++C סימן הפלוס הכפול (++) הוא האופרטור להגדלה עצמית. בדומה לו, בשפת ++C קיים **אופרטור להקטנה עצמית** (Decrement operator), המשמש להקטנת ערך המשתנה באחד. סימנו של האופרטור להקטנה עצמית הוא מינוס כפול (--). הפעלת האופרטור להקטנה עצמית זהה להפעלת האופרטור המקביל, המשמש להגדלה עצמית. האופרטור להקטנה עצמית מופיע בשתי צורות : **אופרטור הקטנה עצמית מוקדמת** (Prefix decrement operator) **ואופרטור הקטנה עצמית מאוחרת** (Postfix decrement operator)**.** בתוכנית הבאה, **DECCOUNT.CPP**, מוצג השימוש באופרטור להקטנה עצמית :

```
#include <iostream.h>
void main(void)
 {
  int small_count = 0;
  int big_count = 1000;

  cout << "small_count is " << small_count << endl;
  cout << "small_count-- yields " << small_count-- << endl;
  cout << "small_count ending value " << small_count << endl;

  cout << "big_count is " << big_count << endl;
  cout << "--big_count yields " << --big_count << endl;
  cout << "big_count ending value " << big_count << endl;
 }
```

לאחר הידור והרצת התוכנית יופיעו על המסך השורות הבאות:

```
C:\>DECCOUNT  <Enter>
small_count is 0
small_count-- yields 0
small_count ending value is -1
big_count is 1000
--big_count yields 999
big_count ending value is 999
```

על פי תוצאות התוכנית לעיל ניתן לקבוע כי האופרטור להקטנה עצמית פועל בצורה זהה לאופרטור להגדלה עצמית, למעט ההבדל שהאחרון מקטין את ערכו של המשתנה באחד.

אופרטורים אחרים

בפרק הנוכחי התמקדנו באופרטורים האריתמטיים הבסיסים ובאופרטורים להגדלה והקטנה עצמית. בשפת ++C קיימים מספר רב של אופרטורים. להלן טבלה המציגה מספר אופרטורים נוספים שבשימוש בשפה.

טבלה 5.2: רשימת אופרטורים בשימוש בתוכניות ++C.

אופרטור	התפקיד
%	מודולו או שארית. מחזיר את שארית התוצאה של פעולת החילוק.
~	אופרטור המשלים לאחד. הופך את ערך הסיביות של משתנה.
&	אופרטור AND. פעולת AND בסיביות בין שני משתנים.
|	אופרטור OR. פעולת OR בסיביות בין שני משתנים
^	אופרטור XOR. פעולת XOR בסיביות בין שני משתנים.
<<	הזזה אריתמטית שמאלה של סיביות לפי מספר הסיביות.
>>	הזזה אריתמטית ימינה של סיביות לפי מספר הסיביות הדרוש.

סדר הקדימות

כשמרכיבים ביטויים אריתמטיים יש לקחת בחשבון את סדר קדימות הביצוע של האופרטורים בביטויים אלה. לדוגמה, לפעולת הכפל יש קדימות על פני פעולת החיבור, ועל כן פעולת הכפל מתבצעת קודם. נבהיר זאת על ידי בחינת הביטוי הבא:

```
result = 5 + 2 * 3;
```

על פי סדר ביצוע הפעולות שבביטוי ניתן לקבל שתי תוצאות שונות:

```
result = 5 + 2 * 3;
       = 5 + 6;
       = 11;
```

```
result = 5 + 2 * 3;
       = 7 * 3;
       = 21;
```

כדי לגרום לאחידות בחישובים, קיים בשפה סדר קדימות קבוע לביצוע של אופרטורים. בטבלה 5.3 מוצגת סדר קדימות של האופרטורים בשפת C++. הטבלה מחולקת לקבוצות, כאשר הקבוצות העליונות הם בסדר קדימות גבוה יותר. הכוונה לכך שפעולות אלו יתבצעו לפני פעולות שבקבוצות תחתונות יותר. בתוך הקבוצה האופרטורים מקבלים סדר קדימות שווה. כפי שציינו קודם, ניתן לראות בטבלה שאופרטור הכפל קודם לאופרטור החיבור. בטבלה מוצגים כל האופרטורים שבשפת C++. בהמשך נלמד על פעולות האופרטורים השונים.

טבלה 5.3: סדר קדימות של אופרטורים.

אופרטור	פעילות	דוגמה
::	טווח הכרה	Class name::class_member_name
::	טווח הכרה גלובלי	::variable_name
.	סימון נקודה לאלמנט	object.member_name
->	סימון חץ לאלמנט	pointer->member_name
()	קריאה לפונקציה	function(parameters)
[]	ציון מערך	array[index]
!	NOT לוגי	!expression
~	המשלים ל- 1 הופך את הסיבית	~expression
+	חיבור אונרי	+1
-	חיסור אונרי	-1
++	הגדלה עצמית מאוחרת/מוקדמת	variable++ / ++variable
--	הקטנה עצמית מאוחרת/מוקדמת	variable-- / --variable
&	כתובת של	&variable

אופרטור	פעילות	דוגמה
*	מצביע	*pointer
sizeof	גודל טיפוס/אובייקט	sizeof(object) / sizeof(type)
new	אופרטור הקצאת זיכרון	new type
delete	שחרור זיכרון	delete pointer
.*	מציין למצביע	Object.*pointer
->	מציין למצביע	Object->pointer
*	כפל	Expression * expression
/	חילוק	Expression / expression
%	מודולו	Expression % expression
+	חיבור בינארי	Expression + expression
-	חיסור בינארי	Expression - expression
<<	הזזה אריתמטית שמאלה	Expression << expression
>>	הזזה אריתמטית ימינה	Expression >> expression
<	קטן מ	Expression > expression
<=	קטן שווה	Expression <= expression
>	גדול מ	Expression > expression
>=	גדול שווה	Expression >= expression
==	שווה	Expression == expression
!=	שונה	Expression != expression
&	פעולת AND בסיביות	Expression AND expression
^	פעולת XOR בסיביות	Expression XOR expression
|	פעולת OR בסיביות	Expression OR expression
&&	וגם בתנאי	Expression && expression
||	או בתנאי	Expression || expression
?:	תנאי מותנה	(a ? x : y means "if a then x, else y")
=	השמה	Expression = expression+expression
*=	קיצור – כפל	Expression *= expression
/=	קיצור – חילוק	Expression /= expression

אופרטור	פעילות	דוגמה
%=	קיצור – מודולו	Expression %= expression
+=	קיצור – חיבור	Expression += expression
-=	קיצור – חיסור	Expression -= expression
&=	קיצור – פעולת AND בסיביות	Expression &= expression
^=	קיצור – פעולת XOR בסיביות	Expression ^= expression
|=	קיצור – פעולת OR בסיביות	Expression |= expression
<<=	קיצור – הזזה שמאלה	Expression <<= expression
>>=	קיצור הזזה ימינה	Expression >>= expression
,	פסיק	int expression, expression

קביעת סדר הפעולות בביטויים מתמטיים

לעיתים יש צורך לקבוע סדר ביצוע פעולות אריתמטיות שלא על פי סדר הקדימות של השפה. לצורך זה ניתן לשלב בביטויים האריתמטיים - סוגריים, וכך לחלק את הביטוי לקבוצות חישוב נפרדות. מבחינת סדר הביצוע של התוכנית, הביטויים שבתוך הסוגריים יתבצעו ראשונים.

נבהיר זאת על ידי בחינת הביטוי הבא:

```
result = (2 + 3) * (3 + 4);
```

להלן סדר הפעולות לביצוע:

```
result = (2 + 3) * (3 + 4);
       = (5) * (3 + 4);
       =    5 * (7);
       =    5 * 7;
       =    35;
```

על ידי שימוש בסוגריים, מחלקים את הביטויים לקבוצות, וכך קובעים את סדר ביצוע הפעולות האריתמטיות. הדוגמה הקודמת עוסקת במספרים בלבד, אך אפשר להקיף בסוגריים גם משתנים, או צירוף של משתנים ומספרים:

```
cost = (price_a + price_b) * 1.06;
```

פעולות חשבוניות עלולות לגרום לשגיאות גלישה (Overflow)

בפרק 4 למדנו, כי כאשר מקצים למשתנה ערך שאינו נמצא בתחום הערכים שטיפוס המשתנה לאחסן, נוצרת שגיאת גלישה. באותו אופן, עלולות גם פעולות חשבוניות לגרום שגיאות גלישה: כאשר מקצים למשתנה ערך שנובע מפעולה חשבונית כלשהי, והערך **חורג** מטווח הערכים של טיפוס המשתנה הזה. לדוגמה, התוכנית **MATHOVER.CPP** ממחישה את הנושא. שים לב להסבר שאחריה.

```
#include <iostream.h>

void main(void)
  {
     int result;
     result = 200 * 300;
     cout << "200 * 300 = " << result << endl;
  }
```

כפי שניתן לראות, מוקצית מכפלת הערכים 200 ו- 300 למשתנה מטיפוס int. כל אחד מהמספרים נמצא בתחום הערכים המתאימים למשתנה מטיפוס זה (32,768- עד 32,767), אך מכפלתם (60,000) חורגת מהערך המקסימלי שהוא יכול להכיל, ולכן נגרמת שגיאת גלישה.

הידור והרצת התוכנית MATHOVER.CPP יביאו להצגת הפלט הבא על המסך :

```
C:\>MATHOVER <Enter>
200 * 300 =  -5536
```

סיכום

בפרק זה הצגנו כיצד מבצעים פעולות אריתמטיות בשפת ++C. התמקדנו באופרטורים הבסיסיים והאופרטורים להגדלה/הקטנה עצמית, והסברנו מה הוא סדר קדימות בביצוע ביטויים אריתמטיים.

לפני שנעבור לפרק הבא, מומלץ לבחון אם מובנים הנושאים הבאים :

- ✓ בשפת ++C משתמשים באופרטורים +, -, * ו- / לביצוע פעולות חיבור, חיסור, כפל וחילוק, בהתאמה.
- ✓ אופרטור להגדלה עצמית של ערך משתנה, יכול להופיע בשתי גרסאות שונות: הגדלה עצמית מוקדמת והגדלה עצמית מאוחרת (אחרי).
- ✓ אופרטור להקטנה עצמית של ערך המשתנה מופיע גם כן בשתי גרסאות : **הקטנה לפני** ו- **הקטנה אחרי**.

- ✓ בגירסה קדם הגדלה/הקטנה עצמית משתמשים בערך המשתנה **לאחר** שגדל, או קטן ערכו באחד. לעומת זה בגירסה הגדלה/הקטנה עצמית מאוחרת משתמשים בערך המשתנה **לפני** שגדל או קטן ערכו באחד.
- ✓ כדי להגיע לביצוע עקבי של פעולות אריתמטיות, קיים בשפת ++C **סדר קדימות** לביצוע האופרטורים השונים. סדר הביצוע של האופרטורים בתוך ביטוי הוא מימין לשמאל.
- ✓ משתמשים בסוגריים כדי לשלוט על סדר ביצוע הפעולות בכלל, עם אפשרות להכתיב סדר פעולות רצוי כלשהו. בשימוש בסוגריים נוהגים לפי כללי המתמטיקה, האומרים שביצוע פעולות בין סוגריים פנימיים **קודמים** לביצוע פעולות בין הסוגריים המקיפים אותם. **שים לב** שמשתמשים בסוגריים רגילים בלבד!

בפרק הבא נציג את ערוץ הקלט cin. משתמשים בו לקריאת נתוני הקלט המגיעים מהמקלדת והקצאתם במשתנים השונים בתוכנית.

תרגילים

1. כיתבו תוכנית המדגימה את פעולת האופרטור >> באופן הבא:
יש לאתחל משתנה שלם בשם i לערך 5, ולהציג בו את הערכים - i, i<<1, i<<2.

2. כיתבו תוכנית המחשבת את שורשיה של המשוואה ax^2+bx+c. התוכנית תגדיר שלושה משתנים מסוג float: a, b ו- c, ותאתחל אותם לערכים כלשהם לחישוב המשוואה. התוכנית תגדיר את שני שורשי המשוואה, result1 ו- result2, מטיפוס float. השורשים יחושבו על פי הנוסחה הבאה:

```
result1= (-b +√b²-4ac) /2a
result2= (-b -√b²-4ac) /2a
```

יש להדפיס את שורשי המשוואה הריבועית result1 ו-result2.

הערה: *התוכנית צריכה להיעזר בפונקציית הספריה sqrt. חישוב השורש של ערך X מתבצע על ידי הקריאה לפונקציה sqrt(X). הפונקציה שייכת לספריה math.h.*

3. מהו הפלט המתקבל מהרצת התוכנית הבאה:

```
void main(void)
  {
      int y = 1, x;

      x=y++;
      cout << "x=" << x << " y=" << y << endl;

      x=++y;
      cout << "x=" << x << " y=" << y << endl;

      (x=y)++;
      cout << "x=" << x << " y=" << y << endl;

      ++x=y;
      cout << "x=" << x << " y=" << y << endl;
  }
```

4. כיתבו תוכנית אשר תיחשב את ההיקף והשטח של שני מעגלים שהרדיוסים שלהם הינם 4 ו- 5 בהתאמה.

פרק 6

קליטת נתונים מהמקלדת

עד כה למדנו על ערוץ הפלט cout המשמש להצגת נתונים וחישובים על המסך. בפרק זה נכיר את ערוץ הקלט cin, המשמש לקליטת נתונים מהמקלדת. כמו כן נראה כיצד מאחסנים את הנתונים הנקלטים בתוך משתנה אחד או יותר.

בפרק זה נדון בנושאים הבאים:

- ערוץ הקלט cin משמש לקריאת תווים ומספרים שהמשתמש מקליד, והקצאתם למשתנים מסוימים.
- ניתן להשתמש בתוכן של משתנה, אשר הוקצה לו ערך באמצעות cin, כאילו הערך הוקצה לו על ידי אופרטור ההשמה של ++C.
- כאשר מקצים למשתנים ערכים באמצעות ערוץ הקלט cin, צריך להתאים לכל משתנה את הערך המתאים לטיפוס (סוג) שלו, וגם לשים לב לטווח הערכים שהוא יכול לאחסן.

בפעולות הקשורות בהצגת נתונים מפעילים ערוץ הפלט cout על ידי שימוש **באופרטור ההכנסה** << (**Insertion operator**) של הערוץ. בצורה דומה, בפעולות הקשורות בקליטת נתונים מפעילים את ערוץ הקלט על ידי שימוש ב**אופרטור השליפה** >> (**Extraction operator**) של הערוץ.

ערוץ הקלט cin

קליטת נתון מהמקלדת על ידי שימוש בערוץ הקלט מחייב קביעת משתנה שבו יאוחסן ערך הקלט. בתוכנית הבאה, **FIRSTCIN.CPP**, מוצג השימוש בערוץ הקלט cin לקליטת מספר מהמקלדת. בתוכנית זו, המספר שנקלט מאוחסן במשתנה number.

```
#include <iostream.h>

void main(void)
 {
   int number;   // The number read from the keyboard

   cout <<"Type your favorite number and press Enter: ";
   cin >> number;
   cout << "Your favorite number is " << number << endl;
 }
```

לאחר הידור והרצת התוכנית, מופיעה על המסך הודעת קלט המבקשת להקליד ערך של מספר כלשהו. לאחר הקלדת המספר והקשה על <Enter> התוכנית מאחסנת את המספר במשתנה number. בשלב הבא התוכנית משתמשת בערוץ הפלט cout להצגת הודעה מתאימה על מסך, הכוללת את המספר שנקלט.

התוכנית twonbrs.CPP קולטת שני מספרים. לאחר קליטתם הם מאוחסנים במשתנה first ובמשתנה second, על פי סדר הקליטה. ערכי המשתנים המוצגים על המסך על ידי שימוש בערוץ הפלט cout.

```
#include <iostream.h>

void main(void)
 {
   int first, second;  // Numbers typed at the keyboard

   cout << "Type two numbers and press Enter: ";
   cin >> first >> second;
   cout << "The numbers typed were " << first << "and"
                                     << second << endl;
 }
```

יש לשים לב למבנה משפט הקלט (cin), ולבחון את השימוש בשני אופרטורים לשליפה.

```
cin >> first >> second;
```

במשפט זה, ערוץ הקלט cin מאחסן את ערך הקלט הראשון במשתנה first, ואת ערך הקלט השני במשתנה second.

במקרה שדרושה קליטה של ערך שלישי, ניתן לשרשר למשפט אופרטור שליפה ומשתנה נוספים כמו דוגמה שלהלן.

```
cin >> first >> second >> third;
```

ערוץ הקלט cin מפריד בין המשתנים על ידי הבחנה ב**תו הבלתי נראה** הראשון (**white space character**). תווים, או סימנים בלתי נראים, הם תו-רווח, מקש Tab, או מקש Enter. ניתן לבדוק תכונה זו על ידי הרצה חוזרת של התוכנית twonbrs וקליטת המספרים תוך שימוש בתווים בלתי נראים שונים.

cin וקליטת נתונים מהמקלדת

ניתן להשתמש בערוץ הקלט cin לקליטת נתונים מהמקלדת. השימוש בערוץ הקלט מחייב לקבוע באיזה משתנה יאוחסן הנתון שנקלט. להלן תבנית למשפט קלט תוך שילוב של אופרטור השליפה (<<).

```
cin >> some_variable;
```

תקינות הקלט - שגיאות בקליטה

ערוץ הקלט cin מתאים את הקליטה לטיפוס המשתנה המקבל את הנתון. לכן יש לקחת בחשבון שעלולות להתרחש תקלות בעת הזנת מספרים לא תקינים, או כאשר מזינים מספרים למשתנים שאינם מתאימים. למשל, אם נריץ את התוכנית FIRSTCIN הקודמת ונקליד את הנתון 1000000 כמספר האהוב עלינו, ונקיש Enter, נראה שבהודעת הפלט שהתוכנית צריכה להציג לא יופיע מספר זה. במסע גילוי סיבת הדבר נראה שהמשתנה שעליו מוקצה המספר הוא מטיפוס int, וכידוע משתנה int יכול לקבל את הערכים שבתחום 32,768- עד 32,767 בלבד, והמספר שרשמנו 1000000 הוא מחוץ לתחום הזה. זוהי שגיאת גלישה (Overflow).

אם נשוב ונריץ את התוכנית ונרשום את האותיות ABC כמספר האהוב, שוב **לא** תהיה התאמה בין טיפוס המשתנה לבין הקלט המוצע, שכן אותיות אינן מספרים, וערוץ הקלט מצפה למספר. שגיאה זו היא מסוג **אי התאמה בסוג** (Type mismatch error).

אותם הניסיונות ניתן לערוך בהרצות חוזרות של התוכנית TWONBRS. הקלדה של ערכים שאינם מספרים, ואף הקלדה של מספרים ממשיים (עם נקודה עשרונית) יגרמו לאותו סוג של תקלות שהראינו. בהמשך הספר נלמד לצמצם תקלות אלו ככל האפשר על ידי בדיקות תקינות לקלט.

קליטה של תו אחד

בשתי התוכניות הקודמות ערוץ הקלט מצפה לערכים מטיפוס int. בתוכנית הבאה, **CIN_CHAR.CPP**, מוצג השימוש בערוץ הקלט cin לקליטת תו אחד מהמקלדת. יש לשים לב שהמשתנה הקולט את הנתונים מהמקלדת הוא מטיפוס char.

```
#include <iostream.h>

void main(void)
 {
   char letter;

   cout << "Type any character and press Enter: ";
   cin >> letter;
   cout << "The letter typed was " << letter << endl;
 }
```

לאחר הידור והרצת התוכנית ניתן להבחין שהתוכנית מסוגלת לקלוט תו אחד בכל ריצה. בחלק השני של הספר נציג כיצד קולטים מחרוזות של תווים.

קליטה של מילים מהמקלדת

בחלק השני של הספר נלמד כיצד לאחסן מילים, או אפילו שורה שלמה של טקסט בתוך משתנה. גם נלמד להשתמש בערוץ הקלט cin כדי לקרוא מילים ושורות שלמות. לעת עתה נלמד לכתוב תוכנית הקוראת ערכים מטיפוס float ו-long. בתוכנית **CIN_LONG.CPP** מוצג השימוש בערוץ הקלט cin לקליטת מספר מטיפוס long. באותה דרך ניתן לקלוט מספרים מטיפוסים שונים. המפתח הוא בכתיבה של טיפוס המשתנה במשפט הקלט המשמש לקליטה.

```
#include <iostream.h>

void main(void)
 {
   long value;
   cout << "Type a large number and press Enter: ";
   cin >> value;
   cout << "The number you typed was " << value << endl;
 }
```

ניתוב מחדש של הקלט

כפי שלמדנו בפרק 3, ניתן לנתב את ערוץ הפלט cout לקובץ או למדפסת, במקום למסך. ערוץ הפלט cout מייצג את ערוץ הפלט הסטנדרטי (Standard output) של מערכת ההפעלה. באופן דומה, ערוץ הקלט cin מייצג את ערוץ הקלט הסטנדרטי (Standard input) של מערכת ההפעלה. כתוצאה מכך, ניתן לנתב (Redirect) את הקלט אל התוכנית מקובץ, ולא מהמקלדת. בפרקים הבאים נלמד כיצד לכתוב תוכניות הקוראות ומעבדות קלט מנותב.

סיכום

בפרק זה הצגנו כיצד משתמשים בערוץ הקלט cin לביצוע קליטת נתונים מהמקלדת. לפני שנעבור לפרק הבא מומלץ לבחון אם מובנים הנושאים הבאים:

- ✓ cin הוא ערוץ הקלט בשפת C++, שמאפשר קליטה של נתונים מהמקלדת.
- ✓ השימוש בערוץ הקלט cin מחייב הגדרת המשתנים להקצאת הנתונים הנקלטים.
- ✓ כדי להקצות ערכי קלט למשתנה יש להשתמש ב- cin תוך שילוב האופרטור לשליפת הקלט (>>).
- ✓ במשפטי cin הקולטים מספר ערכים שונים יש צורך להפריד בין הערכים על ידי סימנים בלתי נראים (תו-רווח, Tab ,Enter).
- ✓ במקרה של אי-התאמה בין סוג הערכים הנקלטים לבין טיפוסי המשתנים שערוץ הקלט cin מציב בהם את הערכים האלה, תהיה שגיאה.

בפרק הבא נציג כיצד ניתן להרכיב משפטי תנאי בתוכנית, ונלמד את תחביר הפקודה if בשפת C++.

תרגילים

1. כיתבו מחדש את תרגיל 2 מפרק 4, אלא שהפעם אורך צלע הקוביה יוזן כקלט על ידי המשתמש.

2. כיתבו תוכנית הקולטת שלושה משתנים מהמשתמש (מטיפוס Double) ומדפיסה אותם ואת הממוצע שלהם על המסך. כמו כן, התוכנית תקלוט שלושה תווים ותדפיס את השרשור שלהם (משמע, זה אחר זה ברצף).

3. כיתבו תוכנית הקולטת את המספר 74233 אל משתנה מסוג unsigned int. הדפיסו את התוצאה והבחינו בחריגה (Overflow).

4. יש לכתוב תוכנית המקבלת כקלט את המשכורת לשעה של עובד כלשהו, מספר שעות העבודה בחודש, ואת אחוז המס שיש לנכות מהשכר. התוכנית תדפיס את שכר הנטו של העובד.

פרק 7

הרצת תוכנית על פי תנאים

נחזור להגדרה של תוכנית מחשב: רצף של פקודות הקובעות כיצד על המחשב לפעול כדי לבצע משימה מסוימת. ביצוע התוכניות שהצגנו עד כאן התחיל במשפט הראשון והסתיים במשפט האחרון. סביר להניח שבמהלך ביצוע תוכניות מורכבות יותר, נרצה שקבוצה של משפטים תתבצע כאשר תנאי מסוים מתקיים, ואילו קבוצה של משפטים אחרים תתבצע כאשר תנאי זה אינו מתקיים. במילים אחרות, נרצה שהתוכנית תדע להתנהג לפי תנאי הסביבה בה היא רצה.

בהמשך הפרק נדון בנושאים הבאים:

- השוואת ערכים כדי לקבוע אם הם גדולים, קטנים או שווים לערכים אחרים.
- השימוש והתחביר של משפט if בשפת ++C.
- ביצוע משפטים פשוטים (Simple) ומורכבים (Compound).
- השימוש והתחביר של משפט if-else להרכבת משפטי תנאי, כאשר משפטים מסוימים יתבצעו בתשובה חיובית (True) לתנאי ומשפטים אחרים יתבצעו בתשובה שלילית (False) לתנאי.
- שילוב של מספר משפטי if-else לביצוע בדיקות של חלופות שונות לביצוע משימות בתוכנית.
- השימוש באופרטורים בוליאנים AND (וגם) ו-OR (או) להרכבת משפטי תנאי מורכבים, כגון: האם המשתמש הוא בעל כלב **וגם** האם הכלב הוא דלמטי?

תוכניות היודעות להתנהג על פי תנאי הסביבה מבצעות **עיבוד מותנה** (**Conditional processing**). כלומר, ישנה מערכת התנייה מובנית בתוך התוכנית, המפקחת על הפעלה או אי הפעלה של משפטים מסוימים. בשלב זה, מספר הכלים שהצגנו מאפשרים כתיבת תוכניות שימושיות.

השוואה בין שני ערכים

באופן טבעי, תוכניות המבצעות עיבוד מותנה בודקות את הקיום של תנאים מסוימים. למשל, תוכנית הבודקת אם ציון התלמיד הוא 100; או תוכנית אחרת הבודקת אם עלות המוצר היא מעל 50 ש"ח. כדי לבצע סוג זה של בדיקות או השוואות, בשפת C++ משתמשים באופרטורים המוצגים בטבלה 7.1.

טבלה 7.1: אופרטורי היחס בשפת C++

האופרטור	הבדיקה	דוגמה
==	שני הערכים שווים	score == 100
!=	שני הערכים שונים	old != new
>	הערך השמאלי גדול מהערך השני	cost > 50.00
<	הערך השמאלי קטן מהערך השני	salary < 2,000.00
>=	הערך השמאלי גדול או שווה לערך השני	stock_price>= 30.0
<=	הערך השמאלי קטן או שווה לערך השני	age <= 21

כאשר משתמשים בתוכנית באופרטורי היחס כדי להשוות בין שני ערכים, אפשר לצפות לשתי תוצאות: נכון - true, או לא נכון - false. בכל פקודות if המוצגות בהמשך הספר נשתמש **באופרטורי היחס** (**Relational operators**).

המשפט if

בשפת C++ מאפשר המשפט if לבדוק תנאי מסוים, ולפי התוצאה - לבצע משפט אחד או רצף של משפטים. להלן תבנית משפט if:

```
if (condition_is_true)
   statement;
```

במשפט if מתבצעת בדיקה של תנאי, תוך שימוש באופרטורי היחס. אם התנאי מתקיים (התשובה חיובית), מבוצע המשפט שבהמשך המשפט if. בתוכנית **FIRST_IF.CPP** שלהלן, מוצג השימוש במשפט if. התוכנית משווה בין הציון של התלמיד שבמשתנה TEST_SCORE, לבין הערך 90. אם הציון גדול או שווה לערך 90 התוכנית תציג הודעה על המסך, אחרת, התוכנית תסתיים.

```
#include <iostream.h>

void main(void)
 {
   int test_score = 95;

   if (test_score >= 90)
     cout << "Congratulations, you got an A!" << endl;
 }
```

כצפוי, יש בתוכנית שימוש באופרטור **גדול או שווה** (=<), להשוות בין הציון לבין הערך 90. אם תוצאת ההשוואה היא "נכון", התוכנית מבצעת את המשפט שבהמשך המשפט if. במקרה זה המשפט שנבצע הוא משפט פלט cout, המציג הודעה מתאימה על המסך. לעומת זאת, אם תוצאת ההשוואה היא "לא נכון", התוכנית לא תבצע את המשפט שבהמשך, ולכן לא תוצג כל הודעה על המסך. כדי להבין טוב יותר את המשפט if, מומלץ להריץ שוב את התוכנית ולשנות את הציון לערך קטן מ- 90.

משפטים פשוטים ומשפטים מורכבים

השימוש במשפט if גורר ביצוע של משפטים נוספים בהמשך לבדיקת התנאי. לפעמים מספיק ביצוע של משפט אחד, אך במקרים רבים יש צורך לבצע מספר משפטים. להלן דוגמה למשפט if, הכולל ביצוע של משפט בודד.

משפט פשוט ←

```
if (test_score >= 90)
 cout << "Congratulations, you got an A!" << endl;
```

כאשר דרושה הרצת רצף של משפטים לאחר בדיקת התנאי, חייבים לתחום את קבוצת המשפטים בתוך סוגריים מסולסלים { }. להלן דוגמה למשפט if הכולל ביצוע של מספר משפטים.

משפט מורכב ←

```
if (test_score >= 90)
 {
  cout << "Congratulations, you got an A!" << endl;
  cout << "Your test score was " << test_score<<endl;
 }
```

לסיכום, כדי להריץ מספר משפטים אחרי בדיקת תנאי במשפט if, חייבים לקבץ את המשפטים האלה בין סוגריים מסולסלים { }. התוכנית **COMPOUND.CPP** היא גירסה משופרת של התוכנית הקודמת, ובה מציגים שתי הודעות פלט לאחר בדיקת הציון, אם הוא גדול או שווה לערך 90.

```
#include <iostream.h>

void main(void)
 {
  int test_score = 95;

  if (test_score >= 90)
   {
    cout << "Congratulations, you got an A!" << endl;
    cout << "Your test score was " << test_score << endl;
   }
 }
```

משפטים פשוטים ומורכבים

אוסף המשפטים המתבצע בעת תשובה חיובית של מפעולת התנייה, יכול להיות פשוט - Simple statement (משפט אחד), או מורכב - Compound statement (משפטים אחדים). כאשר התוכנית צריכה לבצע שני משפטים או יותר, המבוססים על התנייה, יש לקבץ את המשפטים האלה בין סוגריים מסולסלים:

```
if  (age >= 18)
  {
    cout << "Make sure you remember to vote!" << endl;
    cout << "Oh yeah, do not forget your ID Card!" << endl;
```

משפטים שיבוצעו אם התנאי אינו מתקיים

בתוכניות הקודמות השתמשנו במשפט if לבדיקת תנאי וביצוע פעולות לפי תוצאת הבדיקה. זאת אומרת, אם התנאי מתקיים (הציון גדול או שווה לערך 90) התוכנית מציגה הודעה מתאימה על המסך. לעומת זאת, אם התנאי אינו מתקיים (הציון קטן מ- 90), התוכנית לא מציגה הודעה על המסך ולמעשה היא מסתיימת. ברוב המקרים נרצה שלאחר ביצוע בדיקת תנאי יבוצע רצף משפטים אם התנאי מתקיים, ורצף אחר של משפטים אם התנאי אינו מתקיים. כדי לבצע משפט אחד או רצף של משפטים כאשר תנאי מסוים אינו מתקיים, צריכים להשתמש במשפט else, לפי התבנית הבאה:

```
if (condition_is_true)
   statement;
else
   statement;
```

בתוכנית **IF_ELSE.CPP** מפעילים את המשפט if לבדיקת ציון התלמיד. אם הציון גדול או שווה לערך 90 ותנאי הבדיקה מתקיים, התוכנית תציג הודעת ברכה על המסך. לעומת זאת, אם התנאי אינו מתקיים, התוכנית תציג על מסך הודעה הממליצה לתלמיד להתאמץ יותר לקראת הבחינה הבאה.

```
#include <iostream.h>

void main(void)
 {
  int test_score = 95;

  if (test_score >= 90)
    cout << "Congratulations, you got an A!" << endl;
  else
    cout << "You need to work harder next time!"<< endl;
 }
```

משפטים מורכבים בעקבות משפט else

כפי שראינו קודם, משתמשים בסוגריים מסולסלים כדי לקבץ מספר משפטים המשמשים למילוי מטרה מסוימת. במשפט else ניתן להשתמש בתכונה זו ולקבוע סדרה של פקודות שיבוצעו במקרה שהתנאי אינו מתקיים. בתוכנית **CMP_ELSE.CPP** משולבות קבוצות של משפטים לאחר משפט if ולאחר משפט else.

```
#include <iostream.h>

void main(void)
 {
  int test_score = 65;

  if (test_score >= 90)
   {
    cout << "Congratulations, you got an A!" << endl;
    cout << "Your test score was" << test_score << endl;
   }
   else
   {
    cout << "You should have worked harder!" << endl;
    cout << "You missed " << 100 - test_score <<" points " << endl;
   }
 }
```

מומלץ להריץ את התוכנית מספר פעמים ולקבוע מדי פעם ערכים התחלתיים שונים למשתנה test_score, וכך לראות את הביצוע כאשר התנאי של משפט if מתקיים וכאשר הוא אינו מתקיים.

בתוכנית **GETSCORE.CPP** משתמשים בערוץ הקלט כדי לקלוט את הציון מהמקלדת. בהמשך, התוכנית משווה את הציון לערך 90 ולפי תוצאת ההשוואה היא מציגה על המסך הודעה מתאימה.

```
#include <iostream.h>

void main(void)
 {
  int test_score;
  cout << "Type in the test score and press Enter: ";
  cin >> test_score;

  if (test_score >= 90)
   {
    cout << "Congratulations, you got an A! " << endl;
    cout << "Your test score was " << test_score << endl;
   }
  else
   {
    cout << "You should have worked harder!" << endl;
    cout << "You missed " << 100 -test_score << " points " << endl;
   }
 }
```

לאחר ההידור ניתן להריץ את התוכנית ולהיווכח שהשילוב של משפטי קלט ומשפטי תנאי מעניק לתוכנית עוצמה רבה.

הבנת תהליך ההתנייה - if-else

תוכניות מורכבות בוחנות תנאים שונים ובהתאם להתקיימות התנאים תבצענה קבוצות שונות של משפטים: אוסף משפטים אחד אם התנאי מתקיים ואוסף משפטים אחר, אם התנאי אינו מתקיים. תהליך התנייה זה מתבצע באמצעות משפט if-else, כפי שנראה להלן:

```
if (condition_is_true)
    statement;
else
    statement;
```

אם התוכנית צריכה לבצע יותר ממשפט אחד כתוצאה מקיום, או אי קיום התנאי, יש לקבץ בסוגריים את קבוצת המשפטים הזו:

```
if  (condition_is_true)
  {
      first_true_statement;
      second_true_statement;
  }
else
  {
      first_false_statement;
      second_false_statement;
  }
```

הזחת משפטים לשיפור קריאות תוכנית

ניתן לראות בתוכניות שהוצגו בפרק זה שהמשפטים שלאחר משפטי if, else וסוגר מסולסל שמאלי, כתובים בהזחה ימינה. לעיצוב זה קוראים **הזחה** (**Indentation**), והוא גורם לכך שהתוכנית תהיה ברורה וקריאה יותר. בנוסף, ההזחה מאפשרת לבחון בקלות את התוכנית ולמקד משפטים שקשורים ביניהם. מבחינת המהדר והשפה אין חשיבות לאופן עריכת התוכנית, אך למתכנת הקורא את התוכנית – בוודאי שכן. להלן דוגמה להזחה:

```
if (test_score >= 90)
 {
 cout << "Congratulations, you got an A!" << endl;
 cout << "Your test score was" << test_score << endl;
 }
else
 {
 cout << "You should have worked harder!" << endl;
 cout << "You missed " << 100 - test_score << " points " << endl;
 }
```

משפטי תנאי מורכבים

משפטי התנאי שהצגנו לפני כן מתייחסים לבדיקת תנאי אחד. נוכל לחשוב על מצבים שונים שבהם נרצה לבחון תנאי מורכב, שכולל מספר תנאים המשולבים זה בזה. לדוגמה, נרצה לבדוק אם התלמיד עונה על שני תנאים: (1) אם הציון שלו במתמטיקה גדול או שווה לערך 90 (2) אם התלמיד מסווג בכיתת המתקדמים. דוגמה אחרת היא, בדיקה אם אדם הוא בעל כלב והאם הכלב הוא דלמטי. לבדיקות אלו ניתן להשתמש ב**אופרטור AND** (**&&**) - **"וגם"** של שפת ++C. במקרים אחרים נרצה לערוך בדיקות מסוג אחר, בהם נרצה לברר אם הנבדק עונה על תנאי אחד מבין התנאים המצוינים. כמו, לדוגמה: אם בעל-החיים הוא כלב או אם הוא חתול. לצורך זה ניתן להשתמש ב**אופרטור OR** (**||**) - **"או"**. שני האופרטורים האלה הם **אופרטורים לוגיים** (**Logical operators**).

כאשר משתמשים באופרטורים לוגיים AND ו- OR לבדיקת מספר תנאים, יש לשים את ביטויי התנאי בתוך סוגריים כדלקמן:

תנאי שלם

```
if ((user_owns_a_dog) && (dog == dalmatian))
```

כפי שמוצג לעיל, כל ביטוי שנבדק מוקף בסוגריים. **משפט התנאי המורכב נמצא כולו בתוך סוגריים**. כדי לקבל תוצאה "אמת" (true) בבדיקה המשתמשת באופרטור AND (&&), **כל** התנאים המופיעים במשפט **חייבים להתקיים**. אם ביטוי אחד בתוך המשפט אינו **מתקיים** (אינו "אמת"), אז התשובה לבדיקה תהיה "לא נכון" או "שקר" (false). למשל, במשפט התנאי "האם אדם מסוים הוא בעל כלב וגם האם הכלב הוא מסוג דלמטי" מספיק שהאדם אינו בעל כלב, כדי שהתנאי כולו לא יתקיים. באופן דומה, אם הכלב אינו מסוג דלמטי, לא משנה שאדם הוא בעל כלב - התנאי כולו לא מתקיים. כדי שהתנאי אכן יתקיים, חייב להיות לאדם כלב וגם הזן של הכלב חייב להיות מסוג דלמטי.

במשפט הבא משתמשים באופרטור OR (||) כדי לקבוע אם האדם הוא בעלים של כלב, או של חתול.

```
if ((user_owns_a_dog) || (user_owns_a_cat))
```

כדי לקבל תוצאת "אמת" בבדיקה המשתמשת באופרטור OR (||) מספיק שתנאי **אחד מתוך התנאים** המופיעים במשפט מתקיים. למשל, במשפט הקודם אם האדם הוא בעל כלב, הביטוי יהיה "אמת", ואם הוא בעל חתול בלבד, הביטוי יהיה גם-כן "אמת". כמובן, שאם אדם הוא גם בעל כלב וגם בעל חתול, הביטוי גם כן מתקיים. המצב היחיד שהביטוי לא מתקיים הוא אם האדם אינו בעל כלב ואינו בעל חתול, או במילים אחרות, אף לא אחד מתנאי המשפט מתקיים.

ערכים מספריים לתוצאות אמת ושקר

שפת ++C מציגה את הערך הלוגי "אמת" בשימוש בכל מספר מלבד 0, ואת הערך הלוגי "שקר" באמצעות 0 בלבד. התייחסות זו של השפה נוחה מאוד לניסוחם הקריא של תוכניות. למשל, user_owns_a_dog הוא משתנה באמצעותו נאבחן אם אדם הוא בעל כלב או לא. אם אדם אינו בעל כלב ערך המשתנה יהיה 0:

```
user_owns_a_dog = 0;
```

אם האדם הוא בעל כלב, ערך המשתנה יהיה כל מספר השונה מ- 0, לדוגמה - 1:

```
user_owns_a_dog = 1;
```

התוכנית יכולה להשתמש בערך של המשתנה בתוך משפט התנאי if, כמו שנראה כאן:

```
if (user_owns_a_dog)
```

אם ערך המשתנה שונה מאפס, תתקבל תוצאת הבדיקה כ-"אמת", והתנאי יתקיים. לעומת זאת, אם ערך המשתנה שווה לאפס, תוצאת הבדיקה היא "שקר", והתנאי אינו מתקיים. כדי לנצל את היכולת של ++C להציג מצבי "אמת" או "שקר", נציג משפט תנאי אשר שווה למשפט הקודם:

```
if (user_owns_a_dog !=0)
```

בתוכנית **DOG_CAT.CPP** הפקודה if משתמשת במשתנים user_owns_a_dog ו- user_owns_a_cat לקביעת סוג בעל החיים שבבעלות האדם.

```
#include <iostream.h>

void main(void)
 {
   int user_owns_a_dog = 1;
   int user_owns_a_cat = 0;

   if (user_owns_a_dog)
     cout << "Dogs are great" << endl;

   if (user_owns_a_cat)
     cout << "Cats are great" << endl;

   if ((user_owns_a_dog) && (user_owns_a_cat))
     cout << "Dogs and cats can get along" << endl;

   if ((user_owns_a_dog) || (user_owns_a_cat))
     cout << "Pets are great!" << endl;
 }
```

כדי להבין טוב יותר את המשפט, מומלץ לבצע הרצות חוזרות של התוכנית ולשנות בכל הרצה את ערכי המשתנים. פעם להציב 0 לשני המשתנים, ופעם אחרת להציב 1 לשניהם. אפשר להריץ פעם נוספת כאשר ערכי המשתנים יהיו 0 ו-1 לסירוגין. יש לשים לב לפשטות התכנות ולקלות ההפעלה של האופרטורים הלוגיים AND ו- OR לכתיבת משפט בדיקה של תנאים מורכבים.

האופרטור NOT

ניתן להתנות ביצוע של פעולה מסוימת ב״אמת״ של תנאי מסוים או ב״שקר״ שלו. נכונות התנאי לפני הבדיקה של משפט if יכולה להשתנות על ידי האופרטור **NOT** (**!**), וכך התנאי של if יקבל משמעות אחרת. לדוגמה, המשפט הבא בודק אם האדם אינו בעל כלב:

```
if (! user_owns_a_dog)
  cout << "You should buy a dog" << endl;
```

תפקיד האופרטור NOT (לא) הוא להפוך את משמעות התנאי מ-״אמת״ ל-״שקר״ ולהיפך, מ-״שקר״ ל-״אמת״. למשל, במידה ואדם אינו בעל כלב, והערך שבמשתנה user_owns_a_dog הוא אפס. כאשר מפעילים על משתנה זה את האופרטור NOT, יהפוך ערך המשתנה מאפס לאחד. במצב זה הערך שייבדק על ידי משפט התנאי יהיה ״נכון״, כי נכון שהאדם אינו בעל כלב, ולכן המשפט הסמוך לפקודה if מתבצע. המשמעות היא שנכונה העובדה שלאדם זה אין כלב. על אף המסורבלות לכאורה של ניסוח זה, הוא מתברר כנוח ובהיר יותר בתוכניות מורכבות. בתוכנית **USE_NOT.CPP** מוצג כיצד משתמשים באופרטור NOT:

```
#include <iostream.h>

void main(void)
 {
   int user_owns_a_dog = 0;
   int user_owns_a_cat = 1;

   if (! user_owns_a_dog)
     cout << "You should buy a dog" << endl;

   if (! user_owns_a_cat)
     cout << "You should buy a cat" << endl;
 }
```

כדי להבין טוב יותר את המשפט ולבחון את תהליך הביצוע של התוכנית, מומלץ לבצע הרצות חוזרות של התוכנית ולשנות את ערכי המשתנים user_owns_a_dog ו- user_owns_a_cat.

הערה



בדיקה של תנאי שלילי (NOT) מורכבת יותר מבחינתנו מאשר בדיקה של תנאי חיובי. עלינו לזכור כי "לא-לא" פירושו "כן". מומלץ לבדוק היטב את התשובה ולנסות עם מספר ערכים שונים כדי להיות בטוחים שהתנאי נכתב כראוי.

אופרטורים לוגיים

קיימים משפטי התנייה שבהם התנאי מורכב מחלקים אחדים. דוגמה לכך הוא משפט תנאי של תוכנית, אשר בודקת אם תנאי השכר של עובד מסוים הם לפי שעות, ואם עובד זה עבד מעל 40 שעות בשבוע שחלף. כאשר קיום תנאי מסוים תלוי בקיום כל תת-חלקיו (בדוגמה זו - שניים), יש להשתמש באופרטור AND (וגם - &&).

כאשר משתמשים באופרטור AND צריך להקיף כל תנאי בסוגריים, וגם לקבץ את כל התנאים יחד בתוך סוגריים נוספים, כפי שמוצג בדוגמה זו:

```
if ((employee_pay == hourly) && (employee_hours > 40)) statement;
```

כאשר קיום משפט התנאי תלוי בקיום אחד החלקים בלבד, יש להשתמש באופרטור OR (או - ||) . גם כאן צריך להקיף בסוגריים כל תנאי וגם להקיף בסוגריים נוספים את כל התנאים.

הדוגמה הבאה מדגימה את השימוש באופרטור OR על ידי בדיקה, אם המשתמש הוא בעליה של מכונית, או בעליו של אופנוע:

```
if ((vehicle == car)  ||  (vehicle == motorcycle)) statement;
```

בדוגמאות הקודמות התייחסנו למצב שבו התנאי, או קבוצת התנאים, מתקיימים. כאשר רוצים שהתוכנית תבצע משפט, או קבוצת משפטים, כאשר התנאי אינו מתקיים. לשם כך צריך להשתמש באופרטור השלילה NOT (לא - !). אופרטור זה הופך תנאי אמת ("מתקיים") לתנאי שקרי ("לא מתקיים"), ולהיפך. האופרטורים AND, OR ו-NOT הם אופרטורים לוגיים (logical operators).

בדיקת תנאים שונים

בתוכניות הקודמות בפרק זה בוצעו פעולות בהתאם לתוצאת הבדיקה הלוגית של תנאי. בחלק מהן בוצעו פעולות כאשר התנאי מתקיים; באחרות בוצעו פעולות מסוימות כאשר התנאי מתקיים, ופעולות אחרות כאשר התנאי אינו מתקיים. קיימים מצבים בהם מעוניינים לבדוק מספר תנאים הקשורים לאותו נושא. למשל, סיווג רמת התלמיד לפי ציון המבחן. תוכנית כזו צריכה לבדוק לדוגמה, אם הציון גדול או שווה לציונים שונים: 60,70,80,90 וכו'. בתוכנית **SHOWGRAD.CPP** שלהלן אנו מבצעים משימה זו על ידי שימוש במספר משפטי if-else.

```
#include <iostream.h>

void main(void)
 {
   int test_score;

   cout << "Type in your test score and press Enter: ";
   cin >> test_score;

   if (test_score >= 90)
     cout << "You got an A!" << endl;
   else if (test_score >= 80)
     cout << "You got a B!" << endl;
   else if (test_score >= 70)
     cout << "You got a C" << endl;
   else if (test_score >= 60)
     cout << "Your grade was a D" << endl;
   else
     cout << "You failed the test" << endl;
 }
```

הפקודה הראשונה בתוכנית בודקת אם הציון גדול או שווה לערך 90. אם התנאי מתקיים, התוכנית מציגה על המסך הודעה מתאימה. לעומת זאת, אם התנאי אינו מתקיים, המשפט "...else if" שבהמשך מבצע בדיקה אם הציון גדול או שווה לערך 80. כאשר התנאי מתקיים התוכנית מציגה על המסך הודעה מתאימה. אם התנאי אינו מתקיים, התוכנית בודקת שוב את הציון לפי ערך נמוך יותר. תהליך הבדיקה חוזר על עצמו עד לסיום הבדיקות של הערכים הרצויים.

משפט switch

כפי שלמדנו, על ידי שילוב של סדרת משפטי if-else ניתן לבדוק משפטי תנאי מרובים. בתוכנית הקודמת שימשו אותנו משפטי if-else בקביעת תחום הערכים שבו נמצא ציון המבחן. במקרים בהם צריך לבדוק שוויון לערכים מסוימים, משתמשים במשפטי **switch** (מיתוג).

כאשר משתמשים במשפט switch, יש לציין ביטוי ולאחריו אפשרות אחת או יותר, שהתוכנית תוכל לבדוק ולראות אם הן מתאימות לביטוי. התוכנית הבאה, **SWITCH.CPP**, מדגימה שימוש במשפט switch שבעזרתו היא מציגה הודעה אשר מתבססת על הציון של הסטודנט.

```
#include <iostream.h>

void main(void)
 {
  char grade = 'B';
  switch (grade) {
   case 'A': cout << "congratulations on your A" << endl;
         break;
   case 'B': cout << "Not bad, a B is ok" << endl;
         break;
   case 'C': cout << "C's are only average" << endl;
         break;
   case 'D': cout << "D's are terrible" << endl;
         break;
   default:  cout << "No excuses! Study harder!" << endl;
         break;
  }
}
```

משפט switch מורכב משני חלקים:

- משפט התנאי שבא אחרי המילה switch.
- המקרים אשר עשויים לקיים את משפט התנאי.

כשתוכנית נתקלת במשפט switch, היא מעריכה את התנאי, ולאחר מכן מחפשת (לפי הסדר) את המקרה, אשר ערכו זהה לערך התנאי. כשנמצא ערך כזה, התוכנית מריצה את אוסף המשפטים המתאים לו. לדוגמה, בתוכנית זו, המקרה של הערך 'B' מתאים לערך התנאי, ולכן מוצגת ההודעה "Not bad, a B is ok". משפט default הוא מקרה **ברירת-המחדל**, אשר מתאים לכל ערך שאינו ברשימת הערכים. על כן, המשפטים שאחריו מתבצעים רק כאשר כל המקרים הקודמים אינם מתאימים לערך התנאי.

שים ♥! כאשר משתמשים במשפט switch, צריך לכלול בו משפטי break לאחר כל משפט case. כאשר התוכנית מוצאת מקרה מתאים מאלה שמוצגים במשפט switch, היא מחשיבה גם את כל המקרים הבאים אחריו כמתאימים, וכתוצאה - היא תבצע את המשפטים הנלווים להם. המושג גלישה מתאר מצב זה וכדי שהתוכנית תבצע **רק** את המשפטים המתאימים למקרה הראשון שעונה על התנאי, צריך להשתמש במשפט break. משפט break מציין לתוכנית **להפסיק** את פעולת משפט switch **ולעבור** למשפט הבא אחריו. בדוגמה שלנו, התוכנית תבצע את המשפט שלאחר "case 'B'" בלבד, ותסיים. אם תסיר את משפטי break מהתוכנית, היא תדפיס במקרה זה גם את המשפטים האחרים, עד לסוף משפט switch.

סיכום

בפרק זה הצגנו כיצד משתמשים במשפט if וכיצד מרכיבים משפטי תנאי מורכבים בתוכנית. כפי שלמדת, התוכנית שלך יכולה להשתמש במשפט if, כדי לבצע קבוצת משפטים אחת כאשר התנאי מתמלא (true), ונשתמש במשפט else, כדי לפרט משפטים שהתוכנית תבצע כאשר התנאי אינו מתמלא (false).

בפרק 8 נלמד כיצד מבצעים משפטי תוכנית באופן מחזורי מספר פעמים רצוי, או עד אשר תנאי מסוים מתמלא. לדוגמה, תוכל לחזור 100 פעמים על קבוצת משפטים כדי לסכם ציונים של 100 סטודנטים, או עד שמקבלים למשל, ערך שלילי בקלט.

לפני שנעבור לפרק הבא, נבחן אם מובנים הנושאים הבאים:

- ✓ בשפת ++C אופרטורי היחס מאפשרים השוואה בין שני ערכים; אם שני ערכים שווים, או לא שווים, אם ערך אחד גדול או קטן מהשני.
- ✓ המשפט if מאפשר התניית ביצוע פעולות בהתאם להגדרות של מחבר התוכנית.
- ✓ המשפט else מאפשר ביצוע פקודות אחרות אחרי if, בהתאם להגדרת מחבר התוכנית.
- ✓ בשפת ++C כל ערך שאינו אפס מייצג את הביטוי "נכון". לעומת זאת, ערך אפס מייצג את הביטוי "לא נכון".
- ✓ האופרטורים הלוגיים AND (&&) ו- OR (||) מאפשרים להרכיב משפטי תנאי מורכבים, המבצעים בדיקה של מספר תנאים באותו המשפט.
- ✓ האופרטור הלוגי NOT (!) מאפשר בדיקה ישירה של תנאים מהסוג "לא נכון".
- ✓ במקרים בהם צריך לבדוק שוויון בין תנאי מסוים לבין אוסף ערכים מסוימים, משתמשים במשפט switch.
- ✓ כאשר התוכנית מוצאת מקרה המתאים לאחד התנאים שבמשפט switch, היא מחשיבה גם את כל המקרים הבאים אחריו כמתאימים. משפט break מציין לתוכנית להפסיק את פעולת משפט switch ולעבור למשפט הבא אחריו.

תרגילים

1. כיתבו תוכנית המקבלת כקלט שלושה מספרים. אם כולם שווים, יש להדפיס:

```
All numbers are equal
```

אחרת, יש להדפיס:

```
These numbers are not equal
```

2. כיתבו תוכנית הקולטת מספר. אם ערכו גדול מ- 500 או קטן מהמספר 500-, התוכנית מדפיסה 'a', אחרת היא מדפיסה 'b'.

3. כיתבו תוכנית המקבלת שני משתנים בוליאנים, בשמות X ו-Y, ומחשבת את הפונקציה XOR. הפונקציה XOR מוגדרת באופן הבא:

ערך X	**ערך Y**	**X XOR Y**
TRUE	TRUE	FALSE
FALSE	TRUE	TRUE
TRUE	FALSE	TRUE
FALSE	FALSE	FALSE

ערך 0 שקול לערך FALSE, וכל ערך פרט לאפס, שקול לערך TRUE.

הערה*: חישוב הפונקציה יתבצע על ידי משפט if.*

4. כיתבו תוכנית הקולטת שני מספרים ותו המייצג פעולה חשבונית (+, -, *, /). התוכנית תדפיס את תוצאת הפעולה החשבונית בין שני המספרים.

פרק 8

לולאות

בפרק הקודם הצגנו כיצד להשתמש במשפטים if-else כדי לגרום שהתוכנית תבצע פעולות מסוימות לפי תנאים שונים. באותו הקשר, ניתן לצפות שתוכנית תבצע פעולות שונות מספר פעמים, עד למילוי תנאים מסוימים. בפרק זה נלמד כיצד לגרום שהתוכנית תבצע משפט אחד או קבוצה של משפטים עד למילוי תנאי מסוים.

בהמשך הפרק נדון בנושאים הבאים:

- משפט for לביצוע חוזר של קבוצת משפטים מספר פעמים, הנקבע על ידי המתכנת.
- משפט while לביצוע חוזר של קבוצת משפטים מספר פעמים, עד אשר מתמלא תנאי ידוע.
- משפט do-while לביצוע של קבוצת משפטים לפחות פעם אחת, עם אפשרות ביצוע חוזר שלהם עד אשר מתמלא תנאי ידוע.

היכולת לביצוע חוזר של קבוצת משפטים היא אחת התכונות המעניקה לשפות התכנות עוצמה רבה. נסביר זאת על ידי ניתוח של התוכניות המוצגות בהמשך.

ביצוע חוזר של משפט בודד מספר פעמים קבוע

ביצוע חוזר של משפט אחד, או של מספר משפטים, היא אחת הפעולות הנפוצות בתוכניות מחשב. לדוגמה, תוכנית המשתמשת באותה סדרת משפטים כדי להדפיס חמישה עותקים של קובץ מסוים. דוגמה אחרת היא קטע תוכנית שחוזר על עצמו, כדי לחשב את הערך הנוכחי של תיק השקעות, הכולל 30 מניות סחירות בבורסה. בשפת C++, **המשפט for** מאפשר לבצע קבוצה של משפטים מספר פעמים קבוע מראש.

המשפט for הקרוי גם **לולאת for** (**for loop**) נעזר במשתנה מוגדר בלולאה ומתעדכן לפי התקדמות ביצוע הלולאה. זהו "מונה הלולאה", שמאחסן את המספר הסידורי של ביצוע הלולאה. לדוגמה, בלולאת for הבאה, המשתנה count הוא משתנה העזר של הלולאה. הערך שלו משתנה לפי התקדמות הביצוע של הלולאה. במקרה זה הלולאה תתבצע 10 פעמים:

```
for (count = 1; count <= 10; count++)
    statement;
```

בלולאת for ארבעה רכיבים. תפקידם של שלושת הרכיבים הראשונים הוא לנהל ולפקח על ביצוע הלולאה. הרכיב הראשון קובע ערך התחלתי למשתנה, בדוגמה: count=1. ברכיב זה משתמשים פעם אחת בכניסה ללולאה. הרכיב השני משמש לבדיקת התנאי לביצוע המשפט הכתוב אחרי רכיבי הלולאה, בדוגמה התנאי הוא: count <= 10. אם התנאי אינו מתקיים, הלולאה תסתיים והתוכנית תבצע את המשפט שאחרי הלולאה. אם התנאי מתקיים התוכנית תבצע את המשפט הכלול בלולאה ותשתמש ברכיב הרביעי של הלולאה כדי לשנות את הערך של המשתנה הפנימי של הלולאה. בדוגמה זו מקדמים את ערך המשתנה count באחד על ידי המשפט count++. השלב הבא בתוכנית הוא הבדיקה החוזרת של הרכיב השני , תנאי עצירת הלולאה. על פי התוצאה של הבדיקה הזו הלולאה תמשיך לרוץ, או תעצור.

```
for (count = 1; count <= 10; count++)
```

ערך התחלתי **התנאי** **קידום משתנה הלולאה**

בתוכנית **FIRSTFOR.CPP** להלן, מתואר השימוש במשפט for. התוכנית מציגה על המסך את המספרים מ- 1 ועד 100.

```
#include <iostream.h>

void main(void)
 {
   int count;

   for (count = 1; count <= 100; count++)
     cout << count << ' ';
 }
```

בתחילת הביצוע של לולאת for ניתן ערך התחלתי 1 למשתנה count. בהמשך מתבצעת בדיקת התנאי לעצירת הלולאה: אם הערך של המשתנה count קטן, או שווה לערך 100. כאשר התנאי מתקיים, מתבצע המשפט הכלול במשפט התנאי של הלולאה ואילו הערך של המשתנה count מקודם באחד. בשלב הבא נבדק קיום התנאי והפעולות חוזרות על עצמן עד אשר התנאי הנבדק על ידי הרכיב השני אינו מתקיים והלולאה מסתיימת. כדי להבין טוב יותר את המשפט, מומלץ לבצע הרצות חוזרות של התוכנית ולשנות כל פעם את הערך שבתנאי העצירה של הלולאה: 10, 20 ואפילו 500.

בתוכנית **ASKCOUNT.CPP** שיפרנו במעט את התוכנית הקודמת. בתחילת ביצוע התוכנית המשתמש מתבקש להזין את הערך לסיום הלולאה. בהמשך, התוכנית תציג על המסך את המספרים המתחילים ב- 1 ומסתיימים בערך הרצוי.

```
#include <iostream.h>

void main(void)
 {
   int count;
   int ending_value;

   cout << "Type in the ending value and press Enter: ";
   cin >> ending_value;

   for (count = 1; count <= ending_value; count++)
     cout << count << ' ';
 }
```

מומלץ להריץ את התוכנית ולהזין ערכים שונים כגון 10, 1 ואפילו 0. במקרה שנזין את הערך 0, או ערך שלילי כלשהו, הלולאה לא תתבצע מכיון שבתחילת הביצוע התנאי count <= ending_value איננו מתקיים, ואז הלולאה מסתיימת מייד.

לולאת for לביצוע חוזר של קבוצת משפטים

בפרק 7 למדנו שבפקודות if-else המבצעות סדרה של משפטים שונים הקשורים לתוצאות בדיקות התנאי, יש צורך לקבץ את המשפטים האלה בתוך סוגריים מסולסלים { }. אותה דרישה קיימת גם בקשר ללולאה for. בתוכנית הבאה **ADD1_100.CPP** מוצג על המסך בכל סיבוב של הלולאה המספר התורן וסכום של כל המספרים שהוצגו לפניו (זוהי דוגמה, ועל כן אל תחפשו משמעות רבה לפעולה זו).

```
#include <iostream.h>

void main(void)
 {
   int count;
   int total = 0;
   for (count = 1; count <= 100; count++)
     {
       cout << "Adding " << count << " to " << total;
       total = total + count;
       cout << " yields " << total << endl;
     }
 }
```

הקבצת המשפטים הקשורים ביניהם מאפשרת לבצע סדרת פעולות בכל סיבוב, או **איטרציה** (**Iteration**) של הלולאה.

שינוי בקצב קידום הלולאה

בתוכניות הקודמות בפרק זה הצגנו לולאות שקצב הקידום של משתנה העזר המפקח על הקידום הינו ב-1. עלינו לדעת שלולאת for מאפשרת קידום של משתנה העזר שלה בכל ערך שהוא. לדוגמה, בתוכנית **BY_FIVES.CPP** מוצגים המספרים שבין 0 ל- 100 בקפיצות של 5.

```
#include <iostream.h>

void main(void)
 {
   int count;

   for (count = 0; count <= 100; count += 5)
     cout << count << ' ';
 }
```

לאחר הידור והרצת התוכנית יוצגו על המסך המספרים 10,5,0 וכך הלאה עד 100. יש לשים לב להרכב משפט הקידום של משתנה העזר בלולאה:

```
count += 5;
```

לעיתים רוצים לשנות ערך של משתנה (הוספה, למשל) ואחר כך לתת למשתנה את הערך החדש שהתקבל. בשפת ++C קיימות שתי דרכים לעשות זאת. להלן דוגמה לדרך הראשונה. בדוגמה זו אנו מוסיפים 5 לערך הנוכחי של המשתנה count:

```
count = count + 5;
```

דרך שנייה, קצרה יותר, לבצע את אותה פעולה:

```
count  += 5;
```

הדרך השנייה היא הנפוצה בשימוש בלולאות בתוכניות כתובות בשפת C++, מכיון שהיא קצרה יותר וקלה יותר לכתיבה.

ניתן לשנות את הערך של המשתנה הפנימי של הלולאה גם כלפי מטה ולקבוע את הרכיב הרביעי עם אופרטור הפחתה, ואף לתת לו ערכים שליליים.

בתוכנית **CNT_DOWN.CPP** מוצגים על המסך המספרים מ- 100 ועד 1, בסדר הפוך.

```
#include <iostream.h>

void main(void)
 {
   int count;

   for (count = 100; count >= 1; count--)
     cout << count << ' ';
 }
```

במקרה זה, הערך ההתחלתי של המשתנה הוא 100 ובכל סיבוב בלולאה ערך משתנה העזר יורד ב- 1. כאשר הערך מגיע ל- 0, מסתיים ביצוע הלולאה.

הישמר מפני לולאות אינסופיות

לולאת for (for loop) הינה אחד הכלים שבאמצעותם התוכנית יכולה לחזור מספר פעמים רצוי על קבוצת משפטים מוגדרת. משתנה בקרה מונה את מספר החזרות אשר הלולאה ביצעה. כשהתוכנית מגיעה לתנאי הסיום של הלולאה, היא מפסיקה לחזור על המשפטים הכלולים בלולאה, ועוברת למשפט הבא אחרי לולאת for.

לולאות אינסופיות (Infinite loops) נוצרות בשל טעות בתוכנית, שבעקבותיו תנאי הסיום של הלולאה לעולם אינו מתקיים. כתוצאה מכך, הלולאה אינה מפסיקה לחזור על ביצוע המשפטים הכלולים בה, עד אשר עוצרים "בכוח" את התוכנית. לולאת for הבאה היא דוגמה ללולאה אינסופית:

```
for  (count = 0; count < 100; wrong_variable++)
   // Statements
```

המשתנה count הוא משתנה הבקרה של הלולאה, אך הלולאה סופרת את מספר הפעמים שהיא מתבצעת באמצעות משתנה אחר (wrong_variable). על כן, ערכו של המשתנה count לעולם לא יהיה גדול או שווה ל- 100. המשמעות היא שהלולאה תהיה אינסופית. אפשר להיחלץ ממנה רק על ידי הפסקת התוכנית במכוון.

חשוב לציין, כי לולאות for אינן מוגבלות לשימוש במשתנה מטיפוס int כמשתנה בקרה.

התוכנית הבאה, **LOOPVAR.CPP**, מדגימה שימוש במשתנה בקרה מטיפוס char להצגת אותיות האלף-בית בלולאה אחת, ושימוש במשתנה בקרה מטיפוס float - להצגת מספרי נקודה צפה בלולאה שנייה:

```
#include <iostream.h>

void main(void)
 {
   char letter;
   float value;

   for (letter = 'A'; letter <= 'Z' ; letter++)
     cout << letter;

   cout << endl;

   for (value = 0.0; value <= 1.0; value += 0.1)
     cout << value << ' ';

   cout << endl;
 }
```

לאחר הידור והרצת התוכנית יופיע על המסך הפלט הבא:

```
ABDCEFGHIJKLMNOPQRSTUVWXYZ

0  0.1  0.2  0.3  0.4  0.5  0.6  0.7  0.8  0.9 1.0
```

ביצוע לולאה מספר פעמים נתון

אחת הפעולות השכיחות בתוכנית היא חזרה על משפט אחד או יותר, מספר פעמים רצוי. משפט **for** של C++ מאפשר לתוכנית לעשות זאת. משפט for משתמש במשתנה בקרה ששולט על מספר הפעמים של ביצוע הפקודות. תבנית כללית של משפט for היא:

```
for  (initialization; test; increment)
   statement;
```

כאשר הלולאה מתחילה, היא מקצה ערך התחלתי למשתנה הבקרה. אחר כך התוכנית בודקת את תנאי הלולאה. אם התנאי מתקיים (true), התוכנית מבצעת את המשפט הכלול בלולאה. אחר כך התוכנית מקדמת את משתנה הבקרה בערך של ה-increment (במקרה שלנו - 1) וחוזרת לבדוק את התנאי. אם התנאי מתקיים, היא חוזרת על התהליך. אם התנאי אינו מתקיים (false), לולאת for מסתיימת והתוכנית ממשיכה בביצוע המשפט הראשון שאחרי משפט for.

הלולאה while

הלולאה for מאפשרת לבצע הרצה חוזרת של משפט, או קבוצה של משפטים מספר ידוע של פעמים. ישנם מקרים בהם צריך להריץ משפט או קבוצה של משפטים כל עוד מתקיים תנאי מסוים, ללא קביעה מראש של מספר הסיבובים או האיטרציות. למשל, תוכנית הקוראת קובץ (בפרקים מתקדמים יותר נראה כיצד לטפל במידע מאוחסן בקובץ), ומבצעת פעולה כל עוד לא נקלט התו המציין את סוף הקובץ, ולא על פי מספר מוגדר וידוע של פעמים. בשפת ++C מבצעים משימה זו על ידי משפט while. תבנית כללית שלו מוצגת להלן:

```
while (condition_is_true)
    statement;
```

תהליך ביצוע הלולאה מתחיל בבדיקת תנאי הלולאה. כאשר התנאי מתקיים, התוכנית מבצעת את המשפט, או את קבוצת המשפטים שבתוך הלולאה. אחרי ביצוע המשפטים הכלולים בתוך הלולאה נבדק קיום התנאי בשנית. אם התנאי עדיין מתקיים - המשפטים הכלולים בלולאה יבוצעו שוב וחוזר חלילה. רק כאשר התנאי שמציבה הלולאה לא מתקיים, התוכנית ניגשת לביצוע המשפט הבא שמחוץ ללולאה. התוכנית הבאה **GET_YN.CPP** מנחה את המשתמש להקיש על האות Y לתשובה חיובית, או על האות N לתשובה שלילית. לולאת while שבתוכנית מבצעת קליטה של אותיות בודדות מהמקלדת, עד לקבלת האותיות Y או N.

```
#include <iostream.h>

void main(void)
{
 int done = 0;   //Set to true when Y or N is encountered
 char letter;

 while (! done)
  {
   cout << "\nType Y or N and press Enter to continue: ";
   cin >> letter;

   if ((letter == 'Y') || (letter == 'y'))
      done = 1;
   else if ((letter == 'N') || (letter == 'n'))
      done = 1;
   else
      cout << '\a';    // Sound the speaker's bell for
                       // invalid character
   }
 cout << "The letter you typed was " << letter << endl;
}
```

בתוכנית זו ניתן להבחין כי הלולאה while מבצעת מספר משפטים בכל סיבוב. משפטים אלה מקובצים, כנדרש, בתוך סוגריים מסולסלים { }. במקרה זה, הלולאה while נעזרת במשתנה עזר done. כל עוד אין המשתמש מקיש על האות Y (או y) או על האות N (או n), הלולאה ממשיכה להתבצע. כאשר אחת מהאותיות האלו מתקבלת, התוכנית הופכת את ערך המשתנה ל"אמת" והלולאה מסתיימת. לרוב, הלולאה while מופיעה בתוכניות המטפלות בקבצים.

ביצוע לולאה עד אשר תנאי מסוים מתקיים

כשכותבים תוכניות מורכבות, צריך לפעמים לבצע קבוצת משפטים עד אשר תנאי מסוים מתקיים. לדוגמה, תוכנית המחשבת את משכורותיהם של כל העובדים בחברה מסוימת תבצע את הלולאה כל עוד נותר עובד שאת נתוניו צריך לעבד. משפט while משמש, בדרך כלל, כדי לחזור על קבוצת משפטים עד אשר תנאי מסוים מתקיים:

```
while  (condition)
    statement;
```

כאשר תוכנית מגיעה למשפט while היא בוחנת את התנאי ומעריכה אם הוא מתקיים. אם התנאי מתקיים (true), התוכנית מבצעת את משפטי הלולאה. לאחר ביצוע המשפט האחרון בגוף הלולאה, התוכנית בוחנת שוב אם תנאי הלולאה מתקיים, ואם כן, היא מבצעת שוב את משפטי גוף הלולאה. כך היא עושה שוב ושוב, עד שתנאי הלולאה אינו מתקיים (false). במקרה זה התוכנית מפסיקה לבצע את משפטי הלולאה וממשיכה אל המשפט הבא אחרי משפט while.

שגרה של ביצוע משפט פעם ראשונה ללא תנאי: do-while

כפי שלמדנו, בשפת ++C מאפשרת הלולאה while לבצע משפט או קבוצת משפטים, עד למילוי תנאי מסוים. בתחילת הלולאה, נבדק קיומו של תנאי. אם התנאי מתקיים, הלולאה מתבצעת. אם התנאי לא מתקיים, הלולאה לעולם לא תתבצע. ישנם מקרים בהם נצטרך להכין תוכנית שלפי המשימות המוטלות עליה, יש לבצע במהלכה משפט או קבוצה של משפטים לפחות פעם אחת ללא בדיקת קיום תנאי כלשהו - אבל לאחר מכן להתנות את ביצוע המשפטים האלה בקיום תנאי מוגדר מראש. למשימות אלו ניתן להשתמש בלולאה do-while, אשר התבנית הכללית שלה ניתנת להלן:

```
do {
    statements;
} while (condition_is_true);
```

הלולאה do-while מתחילה בביצוע המשפטים שבתוך הלולאה. בסיומם, התוכנית בודקת את תנאי הלולאה. אם התנאי מתקיים, התוכנית תחזור לתחילת הלולאה ותבצע את המשפטים שוב.

```
do {
    statements;
} while (condition_is_true);
```

אם התנאי אינו מתקיים, התוכנית לא תחזור לתחילת הלולאה ותמשיך את ביצוע המשפט הראשון שלאחר הלולאה. שימוש נפוץ בלולאה do-while הוא לשם הצגת תפריטי התוכנית. התוכנית מציגה לפחות פעם אחת את התפריט. במידה והמשתמש בוחר בכל אפשרות מלבד **סיום** (**Quit**), התוכנית תבצע את הפעולה המתבקשת ושוב תציג את התפריט. אם המשתמש בוחר באפשרות **סיום**, הלולאה מסתיימת והתוכנית מבצעת את המשפט הראשון שמחוץ ללולאה.

חזרה על משפטים כל עוד תנאי מסוים מתקיים

כתלות בדרישות התוכנית שלך, ייתכנו מקרים בהם לאחר ביצוע מספר משפטים, לפחות פעם אחת, התוכנית תחזור עליהם כל עוד תנאי מסוים מתקיים. במקרה כזה יש להשתמש במשפט do-while.

```
do {
   statement;
}  while (condition);
```

כאשר תוכנית מגיעה למשפט do-while היא מבצעת מייד את המשפטים הכלולים בגוף הלולאה. לאחר הביצוע, התוכנית בוחנת את קיום תנאי הלולאה. אם התנאי מתקיים (true), היא חוזרת ומבצעת את משפטי הלולאה. אם התנאי שקרי (false), כלומר - אינו מתקיים, התוכנית מפסיקה לבצע את משפטי הלולאה וממשיכה אל המשפט הבא אחרי משפט do-while.

סיכום

תהליך **איטרטיבי/סיבובי** (**Iterative Processing**) מתבטא ביכולת של התוכנית לחזור על ביצוע של משפט אחד או על קבוצת משפטים.

כפי שלמדת, משפט **for** מאפשר לתוכנית לחזור על משפט אחד או יותר מספר פעמים רצוי. משפט **while** מאפשר לחזור על קבוצת משפטים **כל עוד** תנאי מסוים מתקיים. ולבסוף, משפט **do-while** מאפשר לתוכנית לבצע קבוצת משפטים פעם אחת לפחות, וגם לחזור עליהם כל עוד התנאי מתקיים.

לפני שנעבור לפרק הבא, נבחן אם מובנים הנושאים הבאים:

- ✓ הלולאה for בשפת ++C מאפשרת ביצוע חוזר של משפט או של קבוצת משפטים מספר פעמים קבוע.
- ✓ בלולאה for ארבעה רכיבים: ערך התחלתי, תנאי הלולאה, המשפט או קבוצת המשפטים של הלולאה, ופקודה לקידום של משתנה העזר.
- ✓ הלולאה for אינה מחייבת קידום משתנה העזר בערך 1 בלבד, או קידום של משתנה העזר בערכים חיוביים בלבד.
- ✓ הלולאה while בשפת ++C מאפשרת ביצוע חוזר של משפט, או של קבוצת משפטים, כל עוד תנאי הלולאה מתקיים.
- ✓ תוכניות המטפלות בקבצים משתמשות, בדרך-כלל, בלולאה while, מאחר ובאמצעותה ניתן לקרוא את התוכן של הקובץ עד לזיהוי סימן סוף-קובץ.
- ✓ הלולאה do-while מאפשרת לבצע משפט או קבוצת משפטים לפחות פעם אחת, עם אפשרות של חזרה על ביצוע המשפט או קבוצת המשפטים.
- ✓ תוכניות משתמשות, בדרך כלל, בלולאה do-while כדי לעבוד עם תפריטים.
- ✓ בשלוש הלולאות: for, while ו- do-while אי קיום תנאי הלולאה גורם לסיום הביצוע של הלולאה, והמשך התוכנית במשפט הראשון שמופיע אחריה.

בפרק הבא נציג כיצד לחלק תוכנית ארוכה למספר יחידות קטנות ויעילות הנקראות **פונקציות** (**Functions**).

תרגילים

1. כיתבו תוכנית המקבלת כקלט מספר חיובי ומדפיסה למסך את כל המספרים מ-0 עד למספר זה.

2. כיתבו תוכנית המקבלת כקלט מספר חיובי, בדומה לתרגיל הקודם, ומדפיסה את כל המספרים הזוגיים בין 0 למספר זה, כשהם מופרדים בכוכביות.

3. כיתבו תוכנית המדפיסה את לוח הכפל של הספרות 1 עד 10 בצורת טבלה. יש להשתמש בשתי פקודות for מקוננות.

4. יש לכתוב תוכנית המקבלת מספר X, ומדפיסה משולש בעל X שורות מהצורה הבאה. לדוגמה, עבור ערך X=5 נקבל:

```
    *
   ***
  *****
 *******
*********
```

5. כיתבו תוכנית המחשבת ממוצע ציונים של מספר לא מוגבל של תלמידים. התוכנית תקלוט ציונים כל עוד המשתמש לא הזין את הציון 1-. כאשר מתקבל הציון 1-, התוכנית תדפיס את הממוצע ותסיים.

6. כיתבו תוכנית המקבלת כקלט שני מספרים ומדפיסה את כל טווח המספרים בין המספר הגדול לקטן, בסדר יורד. יש להשתמש בפקודה while לביצוע הלולאה.

7. כיתבו תוכנית המבקשת מהמשתמש להזין מספר שאינו שלילי, ובודקת אם המספר הוא מספר ראשוני (אשר מתחלק רק בעצמו וב-1).

חלק 2

השימוש בפונקציות לבניית תוכניות

ככל שהתוכנית הופכת ארוכה ומורכבת, גדל הצורך לחלק אותה ליחידות קטנות יעילות, המעניקות שליטה ומאפשרות תחזוקה קלה. יחידות אלו נקראות פונקציות וכל אחת מבצעת משימה מסוימת. למשל, בתוכנית המטפלת בהנהלת חשבונות ניתן להגדיר **פונקציות (Functions)**. כל אחת מהן מבצעת משימה מסוימת, כמו לדוגמה, קבלת כספים, תשלומים, שכר עבודה וכד׳. על ידי טיפול פרטני בפונקציה הבודדת מקלים על תהליך פיתוח התוכנית השלמה, ומאפשרים להבין את מרכיביה ולשלוט בהם טוב יותר. בנוסף, ניתן לשלב את הפונקציה כיחידה שלמה, גם בתוכניות אחרות, וכך לקצר את זמני הפיתוח של תוכניות חדשות.

חלק זה כולל את הפרקים הבאים:

- ❑ פרק 9 - הפונקציות
- ❑ פרק 10 - שינוי ערכי פרמטרים
- ❑ פרק 11 - יתרונות ספריית ההרצה
- ❑ פרק 12 - משתנים מקומיים וטווח הכרתם
- ❑ פרק 13 - העמסת פונקציות
- ❑ פרק 14 - המשתנה המיוחס בשפת ++C
- ❑ פרק 15 - קביעת ערכי ברירת-מחדל לפרמטרים

פרק 9

הפונקציות

ככל שהתוכנית נעשית ארוכה ומורכבת גדל הצורך לחלק אותה ליחידות קטנות יעילות המאפשרות הקלה בתחזוקה של התוכנית. כל יחידה כזו נקראת **פונקציה** (**Function**) ותפקידה לבצע משימה מוגדרת. תוכנית **קוראת** (**Calls**) **לפונקציה** כאשר נדרש הביצוע של משימה מסוימת במהלך התוכנית. בזמן הקריאה לפונקציה, התוכנית מוסרת לה את הערכים הדרושים לה לביצוע המשימה.

בפרק זה נלמד כיצד ליצור פונקציות בשפת ++C, כיצד להשתמש בהן ונדון בנושאים הבאים :

- פונקציה (Function) היא קבוצת משפטים המבצעים יחדיו מטלה מסוימת.
- קריאה לפונקציה נעשית על ידי כתיבת שמה ואחריו סוגריים, כמו למשל ()beep.
- בסיום פעולתה, יכולה פונקציה להחזיר ערך מטיפוס מסוים, כמו למשל int או float. התוכנית יכולה לבחון ערך זה (בפקודה if, למשל), או להקצות אותו למשתנה.
- הפרמטרים (האקטואליים) שהתוכנית מעבירה לפונקציה (למשל, שם עובד, גיל, או שכר) מקובצים בסוגריים שנכתבים לאחר שם הפונקציה.
- אב טיפוס של פונקציה (Function prototype) משמש להגדרת טיפוס הערך (למעשה, סוג המשתנה) שהפונקציה מחזירה, ואת טיפוסי הפרמטרים שהתוכנית מעבירה לפונקציה.

השימוש בפונקציות חיוני בתהליך כתיבת תוכניות. בשפת ++C קל מאוד ליצור ולהשתמש בפונקציות.

היצירה והשימוש בפונקציה הראשונה

בתוך תוכנית מסוימת, **כל פונקציה צריכה לבצע משימה אחת בלבד**. כאשר נוצר מצב שפונקציה מבצעת יותר ממשימה אחת, כדאי מאוד לחלק אותה למספר פונקציות קטנות לפי מספר המשימות המוגדרות בה. בדומה להגדרת משתנים, גם שם פונקציה בתוכנית אחת חייב להיות ייחודי. בנוסף, מומלץ ששם הפונקציה ישקף את הפעולה שהיא מבצעת. לדוגמה, בטבלה 9.1 ניתנת רשימה של פונקציות, ודי במבט חטוף בשמות אלה, כדי לדעת אילו משימות הן מבצעות.

טבלה 9.1: דוגמאות של שמות בעלי משמעות לפונקציה.

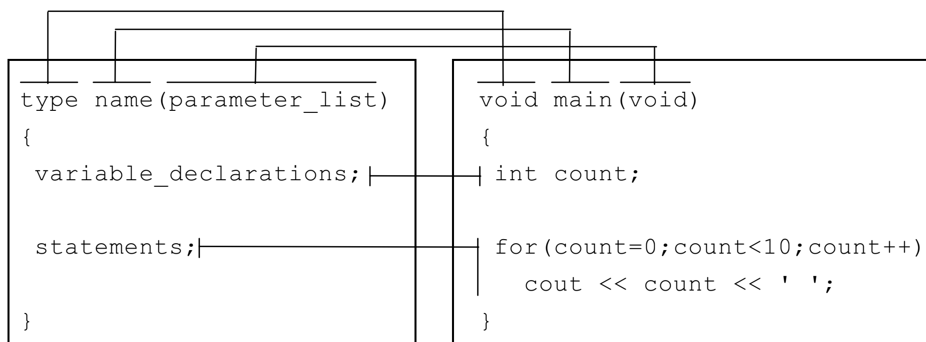
שם הפונקציה	משימה
print_test_scores	הדפסת ציוני בחינה.
accounts_payable	תהליך תשלומים בהנהלת חשבונות.
get_user_name	קבלת שם משתמש.
print_document	הדפסת מסמך מסוים.
calculate_income_tax	חישוב התשלום למס ההכנסה.

המבנה הכללי של הפונקציה דומה למבנה התוכנית הראשית main המופיעה בכל התוכניות המוצגות בספר. הנה תבנית כללית של פונקציה:

```
return_type  function_name(parameter_list)
  {
      variable_declarations;

      statements;
  }
```

כעת נשווה בין המבנה הכללי של הפונקציה לבין מבנה הפונקציה main:



למעשה, התוכנית הראשית מתנהגת כפונקציה לכל דבר. משימת הפונקציה show_message היא להציג הודעה על המסך.

```
void show_message(void)
 {
    cout << "Hello, I've been Rescued by C++" << endl;
 }
```

פירוש המילה void, שבתחילת שם הפונקציה, הוא כפי שהצגנו בפרק 2: הפונקציה אינה מחזירה ערך לאחר סיום הפעולות שהיא מבצעת. לעומת זאת, משמעות המילה void, כאשר היא בתוך סוגריים, היא שהפונקציה אינה מקבלת ערכים מוקדמים (כאשר היא נקראת במהלך ביצוע התוכנית). הפונקציה show_message, משולבת בתוכנית **SHOW_MSG.CPP** שלהלן:

```
#include <iostream.h>

void show_message(void)
 {
   cout << "Hello, I've been Rescued by C++" << endl;
 }

void main(void)
 {
   cout << "About to call the function" << endl;
   show_message();
   cout << "Back from the function" << endl;
 }
```

כאמור, ביצוע תוכנית בשפת ++C מתחיל בפונקציה main. בתוכה ניתן להבחין במשפט הקורא לפונקציה show_message:

```
  show_message();
```

הסוגריים שבהמשך שם הפונקציה מסמנים למהדר ++C שהתוכנית דורשת בשלב זה לקרוא לפונקציה זו. בהמשך הפרק נלמד שבתוך סוגריים אלה ניתן לשלב פרמטרים להעברת מידע לפונקציה. לאחר הידור והרצה, התוכנית תציג על המסך את השורות הבאות:

```
C:\> SHOW_MSG  <Enter>
About to call the function
Hello, I've been Rescued by C++
Back from the function
```

לאחר שהתוכנית פוגשת במשפט קריאה לפונקציה, היא ניגשת לביצוע של המשפטים שבתוך הפונקציה. בסיומם, היא מחזירה את הפיקוד לפונקציה הקוראת, ומבצעת את המשפט הבא לאחר משפט הקריאה לפונקציה.

```
#include <iostream.h>

void show_message(void)
 {
  cout << "Hello, I've been Rescued by C++" << endl;
 }

void main(void)
 {
  cout << "About to call the function" << endl;
  show_message();
  cout << "Back from the function" << endl;
 }
```

התוכנית **TWO_MSGS.CPP** משתמשת בשתי פונקציות show_title ו- show_lessons להצגת שתי הודעות על המסך.

```
#include <iostream.h>

void show_title(void)
 {
  cout << "Book: Rescued by C++" << endl;
 }
void show_lesson(void)
 {
  cout << "Lesson:Getting Started With Functions" << endl;
 }

 void main(void)
 {
   show_title();
   show_lesson();
 }
```

בתחילה, התוכנית קוראת לפונקציה show_title המציגה את ההודעה על מסך באמצעות הפקודה cout. מייד עם סיום הפונקציה show_title התוכנית קוראת לפונקציה show_lesson, אשר באותה דרך מציגה גם היא הודעה על המסך. כאשר הפונקציה show_title מסתיימת, גם התוכנית מסתיימת, מכיון שאחרי הקריאה לפונקציה זו אין משפטים נוספים לביצוע התוכנית הראשית main.

הפונקציות שראינו בפרק זה מבצעות מטלות פשוטות מאד, כך שלמעשה, לא היה לתוכנית צורך אמיתי בפונקציות אלו כדי לבצע את המטלות הכלולות בה. מטרת השימוש בפונקציות בפרק זה היא להראות כיצד **מגדירים פונקציה וכיצד קוראים לה** (מפעילים אותה). פונקציות מסייעות בפישוט תוכניות גדולות ומורכבות על ידי חלוקת התוכנית לחלקים קטנים, פשוטים, חד משמעיים וקלים לשימוש. פונקציות כוללות שורות קוד מעטות בדרך כלל והינן קלות להבנה ולשינוי. אפשר ליצור ספריה (אוסף

של פונקציות שנוהגים להשתמש בהן בתוכניות שונות, וכך לחסוך את זמן הכתיבה והניפוי שלהן שוב ושוב עבור כל תוכנית. כלומר, אפשר להשתמש שוב ושוב בקוד שכבר נכתב, ולא להמציא מחדש את הגלגל...

עוצמתה של שפת ++C (כמו גם של שפת C) נובעת במידה רבה מספריות הפונקציות הנלוות לה ומהיכולת שלה לנצל אותן בדרך יעילה לפיתוח תוכניות מורכבות, ועם זאת - פשוטות לתחזוקה.

הקריאה לפונקציה

התוכנית מבצעת את משפטי הפונקציה על ידי קריאה לפונקציה (Calling a function) ועל כן מכנים אותה גם בשם התוכנית הקוראת (Calling program). כדי לקרוא לפונקציה יש לשלב בתוכנית משפט הכולל את שם הפונקציה וסוגריים, כמו בדוגמה זו:

```
function_name();
```

כאשר דרושה העברת מידע לפונקציה, כותבים את הפרמטרים המעבירים את המידע בתוך הסוגריים, ומפרידים ביניהם בפסיקים. לאחר סיום ביצוע הפונקציה, התוכנית מבצעת את המשפט הראשון הסמוך למשפט הקריאה לפונקציה.

העברת מידע לפונקציות

בשפת ++C ניתן להגביר את יכולת הפונקציה על ידי פרמטרים (מידע) שמעבירים אליה מהתוכנית הקוראת. לכל אחד מהפרמטרים המועברים לפונקציה צריך לציין את הטיפוס שלו, כגון int, או float. לדוגמה, בפונקציה show_number מוגדר פרמטר מטיפוס int.

```
void show_number(int value)

 {
   cout << "The parameter's value is " << value << endl;
 }
```

בעת הקריאה לפונקציה, ניתן להעביר אליה בדרך זו את הערך הרצוי:

```
show_number(1001);
```

ערך שמועבר לפונקציה

שפת ++C ממירה את הפרמטר value בערך המועבר במשפט הקריאה לפונקציה, כפי שמוצג כאן:

```
show_number(1001);

      void show_number (int value)
       {
        cout<<"The Parameter's Value is "<< Value << endl
       }
```

כך מתרגם המהדר את ההעברה לפונקציה:

```
void show_value(1001)
{
cout<<"The parameter's value is <<1001<<endl;
}
```

בתוכנית **USEPARAM.CPP** משתמשים בפונקציה show_number מספר פעמים, כאשר בכל פעם מועבר אליה ערך אחר.

```
#include <iostream.h>
void show_number(int value)
 {
   cout << "The parameter's value is " << value << endl;
 }

void main(void)
 {
   show_number(1);
   show_number(1001);
   show_number(-532);
 }
```

לאחר הידור והרצת התוכנית נקבל על המסך את השורות הבאות:

```
C:\> USEPARAM  <Enter>
The parameter's value is 1
The parameter's value is 1001
The parameter's value is -532
```

ניתן לראות כי כל פעם שהפונקציה נקראת, מועבר למשתנה value הערך הנכון. אם ננסה להעביר לפונקציה ערך מטיפוס שונה מהטיפוס המוגדר לפרמטר, המהדר יציג הודעת שגיאה. לדוגמה, נקבל הודעת שגיאה אם נעביר ערך מטיפוס float לפרמטר מטיפוס int.

ברוב המקרים מועברים מספר ערכים לפונקציה מסוימת. לכל פרמטר מועבר יש צורך להגדיר בפונקציה שם וטיפוס משתנה.

לדוגמה, בתוכנית **BIGSMALL.CPP** הפונקציה show_big_and_little מציגה על המסך את הערך הגדול ואת הערך הקטן שבין שלושת הערכים שמועברים אליה.

```
#include <iostream.h>

void show_big_and_little(int a, int b, int c)
 {
   int small = a;
   int big = a;

   if (b > big)
     big = b;
   if (b < small)
   small = b;
   if (c > big)
     big = c;
   if (c < small)
     small = c;

   cout << "The biggest value is " << big << endl;
   cout << "The smallest value is " << small << endl;
 }

void main(void)
 {
   show_big_and_little(1, 2, 3);
   show_big_and_little(500, 0, -500);
   show_big_and_litlle(1001, 1001, 1001);
 }
```

כאשר התוכנית קוראת לפונקציה, הערכים מועברים כך:

```
show_big_and_little(1,  2,  3);

     void show_big_and_little(int a, int b, int c);
```

לאחר הידור והרצת התוכנית יופיע הפלט הבא על המסך :

```
C:\> BIGSMALL <Enter>
The biggest value is 3
The smallest value is 1
The biggest value is 500
The smallest value is -500
The biggest value is 1001
The smallest value is 1001
```

התוכנית **SHOW_EMP.CPP** משתמשת בפונקציה show_employee כדי להציג על המסך את הגיל (פרמטר מטיפוס int) ואת המשכורת (פרמטר מטיפוס float) של העובד.

```
#include <iostream.h>

void show_employee(int age, float salary)
 {
   cout<<"The employee is " <<age <<" years old" <<endl;
   cout<< "The employee makes $" << salary << endl;
 }

void main(void)
 {
   show_employee(32, 25000.00);
 }
```

ניתן לראות שבפונקציה show_employee מוגדרים הפרמטרים מטיפוס int ומטיפוס float בהתאם לדרישה.

העברת פרמטר לפונקציה

שמות הפרמטרים המוגדרים בפונקציות חייבים להיות ייחודיים. לכל אחד מהם יש לקבוע טיפוס. בעת הקריאה לפונקציה, שפת C++ משבצת לפרמטרים את הערכים המופיעים בתוך הסוגריים של משפט הקריאה לפונקציה. שיבוץ הערכים מתבצע לפי הסדר, משמאל לימין.

החזרת ערכים מפונקציה

הפונקציה מבצעת משימה מסוימת במסגרת התוכנית, וברוב המקרים זוהי משימת חישוב, אשר תוצאותיה מוחזרים לפונקציה הקוראת. על כן, צריך להגדיר את טיפוס הערך המוחזר על ידי הפונקציה. כדי לבצע זאת, די בציון טיפוס הערך המוחזר לפני שם הפונקציה.

לדוגמה, הפונקציה הבאה add_values מקבלת שני פרמטרים מטיפוס int, מחברת אותם, ומחזירה את התוצאה שגם היא מטיפוס int.

```
int add_values(int a, int b)
 {
   int result;

   result = a + b;

   return(result);
 }
```

במקרה זה, הציון int מופיע לפני שם הפונקציה ומגדיר את טיפוס הערך המוחזר על ידי הפונקציה. משפט return מיועד להחזרת הערך מהפונקציה. הוא גורם להחזרת הערך מהפונקציה, מסיים את ביצוע הפונקציה ומעביר את השליטה לתוכנית הקוראת.

את הערך המוחזר ניתן לאחסן במשתנים המוגדרים בתוכנית, כדלקמן:

```
result = add_values(1 , 2);
```

במשפט הקודם התוכנית משבצת למשתנה בשם result את הערך המוחזר על ידי הפונקציה add_values. בצורה דומה ניתן להציג על המסך את הערך המוחזר על ידי הפונקציה, תוך שימוש במשפט cout. להלן דוגמה:

```
cout<< " Sum of values is "<< add_values(500,501) << endl;
```

היישום הקודם של הפונקציה add_values מורכב משלושה משפטים. לפנינו יישום חדש ומקוצר של אותה הפונקציה, שבו מנצלים את תכונותיו של משפט return. בדרך זו אנו ממחישים ומסבירים את מהות הפונקציה:

```
int add_values(int a, int b)
 {
   return(a+b);
 }
```

בתוכנית הבאה **ADDVALUE.CPP** השתמשנו בפונקציה add_values לחיבור מספרים שונים:

```
#include <iostream.h>

int add_values(int a, int b)
 {
   return(a+b);
 }

void main(void)
 {
   cout << "100 + 200 = "  << add_values(100,200) << endl;
   cout << "500 + 501 = " << add_values(500,501) << endl;
   cout << "-1 + 1 = " << add_values(-1, 1) << endl;
 }
```

הפונקציה מחזירה ערך לפי הגדרתה. הפונקציה הקודמת מחזירה ערך מטיפוס int. לעומתה, הפונקציה הבאה average_value מחזירה את הערך הממוצע של שני מספרים מטיפוס int. לרוב, ערך זה יהיה שבר מתמטי כדוגמת המספר 3.5 (מספר שאינו שלם!).

```
float average_value(int a, int b)
 {
   return((a + b) / 2.0);
 }
```

במקרה זה, הציון float שלפני שם הפונקציה מגדיר את טיפוס הערך המוחזר על ידה.

פונקציות שאינן מחזירות ערך

טיפוס הערך המוחזר על ידי פונקציה (למשל, int, float או char) נכתב **לפני** שם הפונקציה. כאשר כותבים את המילה **void** לפני שם הפונקציה, פירוש הדבר שהפונקציה ***אינה מחזירה ערך***.

כדי להחזיר ערך, הפונקציה משתמשת במשפט **return**, אשר משמעותו היא: הפסקת ביצוע הפונקציה והחזרת הערך אל התוכנית הקוראת לפונקציה (שיכולה גם היא להיות פונקציה). במקרה בו טיפוס הפונקציה הוא void, משפט return אינו מחזיר ערך כלשהו, אלא רק מסיים את פעולת הפונקציה. כאשר משפט return אינו מחזיר ערך, הוא נכתב כך:

```
return;
```

בדוגמה הזו, הפונקציה היתה מטיפוס void ועל כן, משפט return אינו עושה דבר, אלא רק מסיים את ביצוע הפונקציה. בפונקציה מטיפוס void אין צורך ב- ;return אלא אם מחליטים לסיים את הפונקציה בשלב כלשהו, לפני הסוף שלה.

הערה



אם בפונקציה כתובים משפטי תוכנית לאחר משפטי return, הם **לא** יבוצעו, מכיון שמשפט return גורם לסיום הפונקציה וחזרה אל התוכנית הקוראת. התוכנית הקוראת ממשיכה בביצוע המשפטים שמייד לאחר משפט הקריאה לפונקציה.

פונקציות מחזירות ערך

כאשר מציבים לפני שם הפונקציה את המילה **void**, הפונקציה **אינה מחזירה ערך**. לעומת זאת, פונקציות המחזירות ערך חייבות לקבל את טיפוס הערך שהן מחזירות, על פי הטיפוס שרשום לפני שם הפונקציה. כדי להחזיר את הערך צריך להשתמש במשפט **return**. כאשר התוכנית מזהה את משפט return, הפונקציה עצמה מסתיימת, ומחזירה את הערך המוגדר לפונקציה הקוראת.

שימוש בערך המוחזר על ידי פונקציה

תוכנית יכולה להשתמש בערך המוחזר על ידי פונקציה באופנים שונים. התוכנית, או הפונקציה **הקוראת** (**The caller**), יכולה להקצות למשתנה את הערך שקיבלה מהפונקציה (הערך המוחזר). היא עושה זאת בעזרת משפט השמה, כפי שנראה להלן:

```
payroll_amount = payroll (employee, hours, salary);
```

המשפט הבא מציג שימוש נוסף בערך המוחזר על ידי פונקציה. לדוגמה, המשפט הבא מציג את הערך המוחזר באמצעות המשפט cout:

```
cout << "The employee made" << payroll (employee, hours, salary) << endl;
```

ניתן להשתמש בערך המוחזר על ידי פונקציה בתוך משפט תנאי, למשל:

```
if (payroll(employee, hours, salary) < 500.00)
          cout << "This employee needs a raise" << endl;
```

הצהרת הפונקציה

כדי שהתוכנית תוכל לקרוא לפונקציה, מהדר שפת C++ צריך לזהות מראש בוודאות את טיפוס הערך המוחזר על ידי הפונקציה. בנוסף, הוא צריך לדעת את מספר הפרמטרים שהפונקציה משתמשת בהם ואת הטיפוס שלהם. בתוכניות שהצגנו בפרק זה, הגדרת הפונקציה שהפעלנו נמצאת באותו קובץ מקור, ו**תמיד** לפני משפט הקריאה לפונקציה. במקרים רבים פונקציות יכולות להופיע במקומות שונים בקובץ המקור (קובץ התוכנית), וגם מקובל מאוד שפונקציה אחת קוראת לפונקציה אחרת.

כדי להבטיח שהמהדר ++C ידע את פרטי הפונקציות השונות שמופעלות בתוכנית, צריך לכתוב בתחילת קובץ המקור של התוכנית את **הצהרת הפונקציה** (**Function declaration**), או כפי שמכנים זאת לעתים: **אב-טיפוס של הפונקציה** (**Function prototypes**). ככלל, הצהרת הפונקציה מספקת מידע אודות **טיפוס הנתון המוחזר מהפונקציה** (**Return type**) ואודות הפרמטרים שלה. המשפטים להלן מציגים הצהרות של פונקציה עבור מספר פונקציות המשמשות אותנו בפרק זה.

```
void show_message(void);

void show_number(int);

void show_employee(int, float);

int add_values(int, int);

float average_value(int, int);
```

ניתן לראות שבכל אב-טיפוס מצוינים טיפוס הערך המוחזר על ידי הפונקציה ומספר וטיפוס הפרמטרים הכלולים בה.

```
 ┌──────────────────────────────────────── Return Type
float average_value(int, int);
                       └──────────────── Parameter types
```

בתוכניות שבהן קוראים לפונקציה שהמהדר לא זיהה את ההצהרה שלה, או את הגדרתה, תופיע הודעה על שגיאת תחביר. ניתן לראות שבקבצי כותר של התוכניות בשפת ++C כתובות ההצהרות של הפונקציות שבשימוש של התוכניות. בתוכנית **PROTO.CPP** שלהלן מוצג השימוש בהצהרות הפונקציות.

```
#include <iostream.h>

float average_value(int, int); //Function prototype

void main(void)
 {
   cout << "The average of 2000 and 2 is " <<
     average_value(2000, 2) << endl;
 }

float average_value(int a, int b)
 {
   return((a + b) / 2.0);
 }
```

בתוכנית זו, הקריאה לפונקציה average_value מופיעה לפני ההגדרה שלה. לכן יש צורך להציב את הצהרת הפונקציה לפני התוכנית-הראשית main, שבה מתבצעת הקריאה לפונקציה. היעדר הצהרת הפונקציה גורם לשגיאת תחביר בשלב ההידור.

אב טיפוס של פונקציה יכול לסייע לך

אב הטיפוס של פונקציה (Function prototype) מציין למהדר את טיפוס הערך המוחזר על ידי הפונקציה, ואת מספר וטיפוסי הפרמטרים שהפונקציה מקבלת. בזמן ההידור המהדר מוודא (בעזרת אב הטיפוס המתאים) שאין טעות בשימוש בערך המוחזר על ידי פונקציה (למשל, שלא הוקצה למשתנה מטיפוס int ערך של פונקציה המחזירה טיפוס (Float), וכי מועברים לפונקציה מספר נכון של פרמטרים מטיפוסים מתאימים. מהדרים ישנים לא ביצעו את הבדיקות הללו, אך מהדרים חדישים מבצעים אותן, וכך הם מקלים רבות על המתכנת בגילוי שגיאות בקוד שכתב.

סיכום

בפרק זה למדנו כיצד משתמשים בפונקציות בשפת ++C. מכיון שהפרק הקיף חומר רב, ייתכן שתפיק תועלת בנסיונות עם תוכניות הדוגמה. לפני שנעבור לפרק הבא מומלץ לבחון אם מובנים הנושאים האלה:

- ✓ ככל שהתוכנית נהפכת לארוכה ומורכבת גדל הצורך לחלק אותה ליחידות קטנות ונוחות לטיפול - אלו הן הפונקציות.
- ✓ לכל פונקציה חייב להיות שם ייחודי.
- ✓ הפונקציות מתוכננות להחזיר ערך לפונקציה הקוראת. הן מקבלות את המידע הדרוש להן באמצעות הפרמטרים של הפונקציה.
- ✓ בפונקציות המחזירות ערך חייבים לציין לפני שם הפונקציה את טיפוס הערך המוחזר על ידן (כמו int, char). לעומתן, בפונקציות שאינן מחזירות ערך יש לציין void לפני שם הפונקציה.
- ✓ בפונקציות המקבלות פרמטרים יש להגדיר לכל פרמטר שם ייחודי וטיפוס הפרמטר. לעומתן, בפונקציות שאינן מקבלות פרמטרים יש לציין void בסוגריים שלאחר שם הפונקציה.
- ✓ מהדר שפת ++C חייב לזהות את טיפוס הערך המוחזר, את המספר וטיפוס הפרמטרים של הפונקציות המשרתים את התוכנית. כאשר הגדרת הפונקציה מופיעה בסוף קובץ המקור לאחר משפט הקריאה אליה, חייבים לכתוב בתחילת קובץ המקור הצהרה של הפונקציה.

בפרק הבא נראה כיצד לשנות את ערכי הפרמטרים המועברים לפונקציות.

תרגילים

1. כיתבו פונקציה המקבלת ערך מטיפוס float, בשם num_to_print, וערך מטיפוס unsigned int, בשם nb_of_times, ומדפיסה את num_to_print, nb_of_times פעמים.

2. כיתבו פונקציה אשר מקבלת ערך מטיפוס float, ומחזירה את הריבוע שלו. השתמשו בפונקציה כדי לחשב את הריבועים של 7, 3.4, -311.3, 4 .

3. כיתבו שתי פונקציות min ו- max. הפונקציה max מקבלת שני ערכים מטיפוס float ומחזירה את הגדול מביניהם. הפונקציה min מחזירה את הקטן מביניהם.

4. כיתבו פונקציה אשר מקבלת שני ערכים, x ו-y, מטיפוס float ומחזירה את x^3+y^3. כיתבו גם פונקציית עזר cube המחזירה את החזקה השלישית של מספר נתון.

5. כיתבו מחדש את תרגיל 6 בפרק 8 והשתמשו בפונקציות min ו-max. שימו לב להערת האזהרה שמייצר המהדר כתוצאה מהצורך להסב ערך מטיפוס int לטיפוס float. שנו את הגדרת min ו-max כדי שלא תתקבלנה הודעות אזהרה.

פרק 10

שינוי ערכי פרמטרים

בפרק הקודם למדנו כיצד ניתן לחלק את התוכנית ליחידות קטנות ויעילות לתחזוקת התוכנית, הנקראות **פונקציות**. כפי שלמדנו, התוכנית יכולה להעביר מידע (פרמטרים) לפונקציות. בדוגמאות שבפרק 9 התוכניות השתמשו או הציגו פרמטרים מבלי לשנות את ערכיהם. בפרק זה נלמד כיצד לשנות את ערך הפרמטרים בתוך הפונקציה. לשינוי ערך של פרמטר בתוך הפונקציה נדרשים מספר צעדים אותם נציג בהמשך.

בפרק זה נדון בנושאים הבאים:

- שינוי ערך של פרמטר פורמלי של פונקציה יכול להתבצע אך ורק אם הפרמטר מועבר כ**מצביע** (Pointer), או כ**משתנה ייחוס** (Reference variable).
- כתובת הפרמטר בזיכרון חייבת להיות ידועה לפונקציה, כדי שזו תוכל לשנות את ערכו.
- כתובת הזיכרון של משתנה נקבעת באמצעות האופרטור **כתובת של** (&).
- האופרטור **מצביע לכתובת בזיכרון** (*), מאפשר לקבוע מהו הערך המאוחסן בכתובת זיכרון נתונה.
- כאשר התוכנית מטילה על פונקציה לשנות ערך של פרמטר שלה, היא מעבירה לה את כתובת הפרמטר בזיכרון המחשב.

שינוי ערך של פרמטר בתוך פונקציה הינו תהליך מקובל מאוד. על כן, ראוי לתרגל את דרך הביצוע באמצעות התוכניות שבפרק זה, כדי להגיע לרמת מיומנות גבוהה.

מדוע בדרך כלל לא ניתן לשנות את ערכי הפרמטרים בתוך הפונקציה?

התוכנית הבאה, **NOCHANGE.CPP**, מעבירה שני פרמטרים לפונקציה display_values. שמות הפרמטרים הם big ו- small. בפונקציה display_values מתבצעת השמה של הערך 1001 לשני הפרמטרים האלה. ואחר כך הפונקציה מציגה את ערכם על המסך. התוכנית מציגה שוב את ערך הפרמטרים על המסך, לאחר סיום ביצוע הפונקציה.

```
#include <iostream.h>

void display_values(int a, int b)
 {
  a = 1001;
  b = 1001;

  cout << "The values within display_values are " << a <<
                                      " and " << b << endl;
 }

void main(void)
 {
  int big = 2002, small = 0;
  cout << "Values before function " << big << " and " <<
                                             small << endl;

  display_values(big, small);

  cout << "Values after function " << big << " and " <<
                                             small << endl;
 }
```

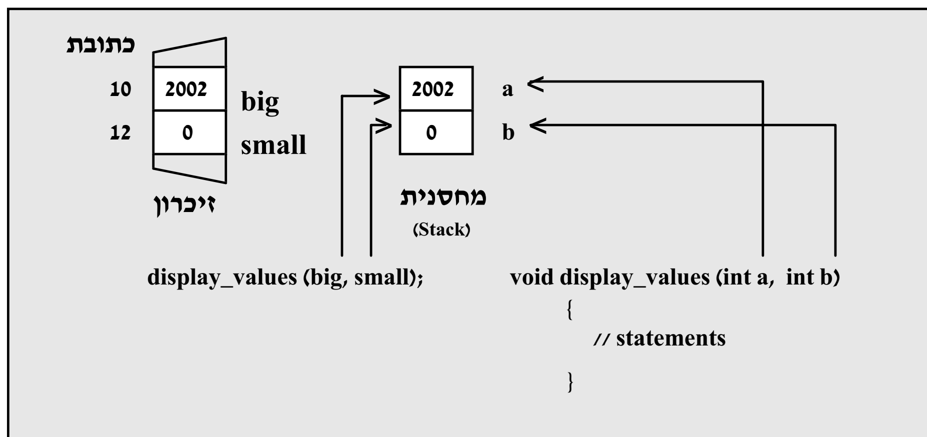
לאחר הידור והרצת התוכנית, תוצגנה על המסך השורות הבאות :

```
C:\>NOCHANGE  <Enter>
Values before function 2002 and 0
The values within display_values are 1001 and 1001
Values after function 2002 and 0
```

בפלט של התוכנית נוכל לראות תופעה מעניינת : בפלט של הפונקציה display_values, ערכי שני הפרמטרים עודכנו ל-1001. אבל לאחר סיום הפונקציה, התוכנית מדפיסה את ערכי הפרמטרים big ו- small, המקוריים כפי שהם מופיעים ב-main, ללא שינוי. כדי להבין את התופעה, בה שינוי ערך הפרמטרים אינו משפיע על הערכים המקוריים של המשתנים big ו- small בפונקציה main, צריך להתבונן היטב במנגנון העברת הפרמטרים לפונקציה בשפת ++C.

המהדר מעתיק את ערכי הפרמטרים אל הפונקציה. הערך המועתק מאוחסן זמנית במקום בזיכרון הנקרא **מחסנית** (**Stack**). הפונקציה משתמשת בערכים המועתקים שבמחסנית לפעולות שהיא מבצעת בתוכה. כאשר הפונקציה מסתיימת, המהדר מוחק את תכולת המחסנית, ואיתה את כל השינויים שהפונקציה ביצעה בערכי הפרמטרים שהועתקו אליה.

כידוע, המשתנה הוא שם של כתובת בזיכרון אשר בו מאוחסן ערך מטיפוס המוגדר למשתנה המסוים בתוכנית. לדוגמה, נניח שכתובת 10 וכתובת 12 בזיכרון המחשב הן הכתובות של המשתנים big ו- small. כאשר התוכנית מעבירה את המשתנים האלה לפונקציה display_values, הם מועתקים אל המחסנית. בתרשים 10.1 מתוארת העברת הפרמטרים לפונקציה dispaly_values, וכיצד היא משתמשת בערכים המועתקים שבמחסנית.



תרשים 10.1: העתקת ערכי הפרמטרים המועברים אל הפונקציה באמצעות המחסנית.

הפונקציה display_values ***אינה*** יכולה לשנות את ערכי המשתנים big ו- small המכילים את הערכים 2002 ו- 0 בהתאמה, מכיון שאין לה גישה לכתובות 10 ו- 12 של הזיכרון, שבהן נמצאים המשתנים המקוריים.

למה פונקציות בשפת ++C אינן יכולות לשנות בדרך כלל את ערכי הפרמטרים

בעת העברת הפרמטרים לפונקציה, מועתקים ערכי הפרמטרים למקום בזיכרון הנקרא **מחסנית** (Stack). כל השינויים שהפונקציה מבצעת בפרמטרים מופעלים במחסנית בלבד. בסיום הפונקציה, נמחקת כל תכולת המחסנית, ואיתה גם כל השינויים שהפונקציה עשתה בערכים המועתקים. מכיון שהפונקציה אינה יודעת היכן כתובות הפרמטרים (המשתנים), היא אינה יכולה לשנות את הערכים שלהם.

שינוי ערכי הפרמטרים

כדי לבצע שינוי בערך פרמטר כלשהו, הפונקציה צריכה לדעת את הכתובת שלו בזיכרון המחשב. כדי להעביר כתובת של פרמטר לפונקציה משתמשים באופרטור "**כתובת של**" (**&**). במשפט הקריאה לפונקציה הבאה מתואר כיצד מעבירים את הכתובת של המשתנים big ו-small לפונקציה change_values.

העברת פרמטרים לפי כתובת

```
change_values(&big, &small);
```

בתוך הפונקציה, במקביל לציון העברת הכתובת של הפרמטרים במשפט הקריאה, צריכים לציין שאכן הכתובת הועברה אליה. לצורך זה מגדירים בפונקציה **משתנה מצביע** (Pointer variable) אחד, או יותר, על ידי צירוף כוכבית **לפני** שם המשתנה. הנה דוגמה:

```
void change_values(int *big, int *small)
```

משתנה מצביע

כדי להציב ערכים למשתנים בפונקציה, צריך לומר למהדר שפועלים עם כתובת של פרמטר. לשם כך צריך להוסיף כוכבית משמאל לשם הפרמטר. בדרך זו ייקבעו השינויים במקום הנכון בזיכרון. להלן דוגמה:

```
*big = 1001;
*small = 1001;
```

בתוכנית **CHGPARAM.CPP** מתואר השימוש בטכניקה שהצגנו לפני כן - שימוש באופרטור כתובת. בתוכנית מעבירים לפונקציה change_values את כתובות הזיכרון של המשתנים big ו-small, המשמשים כפרמטרים לפונקציה. הפונקציה מבצעת השמה בערך המאוחסן בכתובות המשתנים בזיכרון, והדבר גורם גם לשינוי ערך המשתנים המקוריים שבזיכרון, אשר נשמר לאחר סיום ביצוע הפונקציה.

```
#include <iostream.h>

void change_values(int *a, int *b)
 {
   *a = 1001;
   *b = 1001;
   cout << "The values within display_values are " <<
                             *a << " and " << *b << endl;
 }
```

```
void main(void)
 {
   int big = 2002, small = 0;

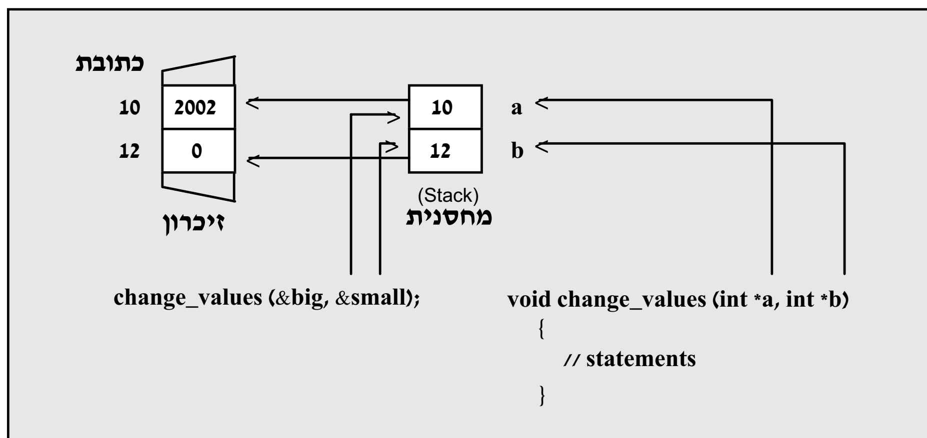
   cout << "Values before function " << big << " and " <<
                                          small << endl;
   change_values(&big, &small);

   cout << "Values after function " << big << " and " <<
                                          small << endl;
 }
```

לאחר הידור והרצת התוכנית, תוצגנה על המסך השורות הבאות:

```
C >\:CHGPARAM   <Enter>
Values before function 2002 and 0
The values within display_values are 1001 and 1001
Values after function 1001 and 1001
```

בפלט התוכנית ניתן לראות שהשינויים שבוצעו בפרמטרים על ידי הפונקציה change_values השפיעו על אותם ערכי המשתנים המוגדרים מחוץ לפונקציה. מכיון שלפונקציה יש גישה למיקום המשתנים בזיכרון, היא יכולה לפעול על הערכים האלה ישירות, ולכן השינויים נשמרים לאחר סיום הפונקציה. בעת העברת הכתובת של פרמטרים לפונקציה, שפת C++ מציבה את הכתובת של כל אחד מהמשתנים במחסנית. תהליך העברת הכתובות של הפרמטרים מתואר בתרשים 10.2.



תרשים 10.2: העברת פרמטרים לפי כתובת.

על ידי השימוש במצביעים (Pointers) לפונקציה change_values נוצרת גישה ישירה לכתובת המשתנים בזיכרון וכך, השינוי בערכם נקבע כנדרש.

שינוי ערכי הפרמטרים בתוך הפונקציה

כדי לשנות ערכים של הפרמטרים, צריך לאפשר לפונקציה ל**גשת לכתובות** של הפרמטרים בזיכרון. כתובות הפרמטרים מועברות לפונקציה על ידי שימוש באופרטור "כתובת של" במשפט הקריאה אליה:

```
some_functions(&some_variable);
```

במקביל להעברת הכתובת במשפט הקריאה לפונקציה, יש לציין בהגדרת הפונקציה עצמה שהפרמטרים שהיא מקבלת הם מטיפוס מצביע (כתובת של משתנה), על ידי צירוף כוכבית לפני שם הפרמטר:

```
void some_function(int *some_variable);
```

בתוך הפונקציה יש לפנות אל המשתנה, תוך צירוף כוכבית לשמו:

```
*some_variable = 1001;
cout << *some_variable;
```

כדי למנוע שגיאות, שפת ++C אינה מתירה לתוכנית להעביר כתובות פרמטרים לפונקציה שאינה מצפה לקבל מצביע כפרמטר.

המהדר גם מפיק הודעת אזהרה כאשר התוכנית מנסה להעביר ערך אל פונקציה שמצפה לקבל כפרמטר ערך של מצביע.

דוגמה נוספת

כאשר התוכנית מעבירה מצביעים לפרמטרים, הפרמטרים יכולים להיות מכל טיפוס שהוא. הפונקציה משתמשת במצביעים מצביעה על המשתנים מהטיפוס המתאים, ומציינת לפני כל אחד מהם כוכבית כדי לציין שהמשתנה הוא מצביע.

בתוכנית **SWAPVALS.CPP** מועברות לפונקציה swap_values כתובות של שני פרמטרים מטיפוס float. תפקיד הפונקציה הוא להחליף בין ערכי הפרמטרים על ידי התייחסות לכתובותיהם בזיכרון.

```
#include <iostream.h>

void swap_values(float *a, float *b)
 {
   float temp;

   temp = *a;
   *a = *b;
   *b = temp;
 }
```

```
void main(void)
 {
   float big = 10000.0;
   float small = 0.00001;

   swap_values(&big, &small);

   cout << "Big contains " << big << endl;
   cout << "Small contains " << small << endl;
 }
```

התוכנית קוראת לפונקציה swap_values, ומעבירה אליה פרמטרים לפי כתובת. תוכנית מתייחסת לפרמטרים על ידי שימוש במצביעים בתוך הפונקציה.

כעת ננתח מספר משפטים חשובים בפונקציה swap_values. ראשית, הפרמטרים המוגדרים a ו-b הם מצביעים לערכים מטיפוס float.

```
void swap_values(float *a, float *b)
```

בתוך הפונקציה מוגדר משתנה temp. טיפוס המשתנה הוא float רגיל (לא מטיפוס מצביע ל- float).

```
float temp;
```

כעת נתבונן במשפט הבא:

```
temp = *a;
```

משפט זה מבצע השמה של הערך בזיכרון שהפרמטר a מצביע עליו (הערך 10000.0 של big) אל משתנה הפונקציה temp. ההשמה תקינה, מכיון שהמשתנה temp גם הוא מטיפוס float. משתנה מטיפוס מצביע הוא משתנה המכיל כתובת בזיכרון המחשב. במשפט הבא מוצהר שהמשתנה temp הוא מצביע של כתובת בזיכרון, שבה מאוחסן ערך מטיפוס float.

```
float *temp;
```

במקרה זה, המשתנה temp יכול להצביע על כתובת בזיכרון שבה מאוחסן מספר ממשי, אבל הוא לא מסוגל לאחסן את הערך עצמו. בנושא "מצביעים" נדון בהרחבה בחלק 3 של הספר. בשלב זה חשוב לדעת, שלצורך העברת פרמטרים לפי ייחוס יש להשתמש במצביעים.

הבנת פעולת המהדר בעזרת תדפיס של שפת אסמבלי

אחת הדרכים הטובות ביותר להבין כיצד מהדר ++C מתייחס למצביעים, היא לבחון את פלט המהדר, אשר מוצג בשפת אסמבלי (Assembly). רוב מהדרי ++C מספקים מתג שורת הרצה (Command line switch), המכוון את המהדר לייצר קוד אסמבלי. על ידי עיון בתדפיס שפת אסמבלי המופק על ידי מהדר ++C, אפשר להבין טוב יותר כיצד המהדר משתמש במחסנית כאשר הוא מעביר פרמטרים לפונקציה.

סיכום

בפרק זה למדנו כיצד לשנות את ערכי פרמטרים בתוך הפונקציה. כדי לעשות זאת, יש להשתמש במצביעים המאפשרים גישה ישירה לכתובת הפרמטר בזיכרון. בפרק 14 נציג טכניקת פרמטרים מיוחסים (References) של שפת ++C, המפשטת את תהליך שינוי ערכי הפרמטרים. הטכניקה שהצגנו בפרק זה נפוצה מאוד בתוכניות הכתובות בשפת C הקלאסית, ולכן חשוב להבין כיצד היא פועלת. לפני שנעבור לפרק הבא, מומלץ לבחון אם מובנים הנושאים הבאים:

- ✓ בצורה רגילה, פונקציה אינה מסוגלת לשנות ערכי פרמטרים.
- ✓ בעת העברת הפרמטרים לפונקציה, שפת ++C מעתיקה את הערכים שלהם למקום בזיכרון הנקרא מחסנית. כל השינויים שהפונקציה מבצעת בפרמטרים מופעלים במחסנית בלבד.
- ✓ כדי לשנות ערכים של פרמטרים, הפונקציה צריכה לדעת את הכתובות שלהם בזיכרון.
- ✓ כתובות הפרמטרים מועברות לפונקציה על ידי שימוש באופרטור "כתובת של" (&).
- ✓ בפונקציה המקבלת כתובת של משתנה, יש להגדיר את הפרמטרים מטיפוס מצביע (כתובת של המשתנה בזיכרון) על ידי צירוף כוכבית לפני שם הפרמטר.
- ✓ בתוך פונקציה כאשר רוצים להשתמש בערך של התא בזיכרון אליו משויך מצביע יש לסמן כוכבית לפני שם הפרמטר.

בפרק הבא נציג כיצד להשתמש בפונקציות של **ספריות ההרצה** (**Run-time libraries**) המסופקות על ידי היצרנים בחבילות המהדרים של שפת ++C. נראה ששימוש בפונקציות אלו עשוי להגביר את עוצמת התוכניות ולחסוך זמן רב בפיתוח שלהן.

תרגילים

1. כיתבו פונקציה בשם sort, המקבלת שלושה משתנים מטיפוס int (by reference), וממיינת אותם מהגדול לקטן.

בשלב ראשון, הקלט יהיה מתוך משתנים בתוכנית. אחר כך תוכלו לכתוב תוכנית דומה, אשר מקבלת את הקלט מהמקלדת.

2. כיתבו פונקציה מהצורה הבאה:

```
int max_divider(int upper_limit, int divider);
```

פונקציה זו מחזירה את המספר השלם הגדול ביותר, שערכו קטן או שווה לערך upper_limit, והוא מתחלק ללא שארית ב- divider. ניתן לבצע את הפעולה ללא הגדרת משתני עזר מקומיים בפונקציה.

אפשר להניח כי divider אף פעם אינו גדול מהערך upper_limit.

3. כיתבו פונקציה func(int *i, int *j) המאתחלת את ערכי המשתנים i ו-j, המתקבלים כקלט מהמשתמש. יש לכתוב שגרת main, אשר תפעיל את הפונקציה func, ותדפיס את ערכי המשתנים.

4. מיצאו את השגיאות בתוכנית הבאה, ותקנו אותה:

```
#include <iostream.h>

int add(int& x, int& y)
  {
      &x= &x + &y;
      return &x;
  }

void main(void)
  {
      int a=1,b=2;
      cout << add(*a, *b);
  }
```

פרק 11

יתרונות ספריית ההרצה (RUN-TIME LIBRARY)

בפרק 9 למדנו כיצד ניתן לחלק את התוכנית לפונקציות שהן יחידות קטנות יעילות לתחזוקת התוכנית והמבצעות משימות מסוימות. אחד היתרונות של הפונקציות הוא שניתן להשתמש בהן בתוכניות שונות.

בפרק זה נציג **פונקציות של ספריות ההרצה הסטנדרטיות** (**Run time libraries**) או בקיצור **פונקציות ספריה**, המסופקות בחבילות המהדרים של שפת ++C. בספריה זו יש אוסף מקיף של פונקציות מוכנות שניתנות לשימוש בכל תוכנית ותוכנית. על ידי ניצול יתרונותיה של ספריית ההרצה מצטמצמת עבודת תכנות הדרושה להשלמת התוכנית. עבודה מייגעת זו מתחלפת בקריאות לפונקציות ספריה, אשר מבצעות את המטלות הדרושות. בהמשך הפרק נדון בנושאים הבאים :

- ❖ השימוש בפונקציות ספריה.
- ❖ הגדרה נכונה של קבצי כותר של ספריות ההרצה.
- ❖ השימוש בהצהרות הפונקציות כדי לדעת אילו פרמטרים יש להעביר לכל פונקציה.

אפשר לנהל יותר מספריית הרצה אחת, כפי שנלמד בהמשך. רוב ספריות ההרצה המסופקות עם המהדרים של השפה מכילות מספר רב של פונקציות שימושיות. השימוש בפונקציות אלו חוסך זמן רב בפיתוח התוכניות ומגביר את עוצמתן. בפרק הנוכחי נראה שהשימוש בפונקציות הספריות פשוט עד מאוד.

השימוש בפונקציות ספריה

בפרק 9 למדנו כי השימוש בפונקציה בתוכנית מותנה בכך שתהיה הגדרה שלה בקובץ המקור שבו היא נקראת, או לחילופין תהיה הצהרה של הפונקציה, או שיוצג אב-טיפוס שלה לפני משפט הקריאה אליה. מכיון שהגדרת פונקציות הספריה נמצאות בספריה עצמה, יש להצהיר עליהן בתוכנית המשתמשת בהן. כדי להקל על מלאכת ההצהרה ולמנוע שגיאות תחביר, מצורף לכל ספריות ההרצה **קובץ כותר** (**Header file**) המכיל את ההצהרות התקינות של פונקציות הספריה שבו. לכן, למתכנת נותר רק להציב בתוכנית את שם קובץ הכותר הנכון במשפט include שבתחילת התוכנית. לדוגמה, בתוכנית הבאה, **SHOWTIME.CPP**, מתואר השימוש בפונקציות הספריה: time ו-ctime להצגת התאריך והשעה הנוכחים על גבי המסך. הצהרת פונקציות ספריה אלו נמצאת בקובץ הכותר time.h.

```
#include <iostream.h>
#include <time.h>        //For run-time library functions

void main(void)
 {
   time_t system_time;

   system_time = time(NULL);

   cout << "The current system time is " <<
     ctime(&system_time) << endl;
 }
```

לאחר הידור והרצת התוכנית יוצגו על המסך התאריך והשעה המצויינים בשעון המערכת.

```
C:\> SHOWTIME  <Enter>
The current system time is Wed Dec 08 16:13:51 1999
```

בתוכנית הקודמת השתמשנו בפונקציות time ו- ctime. התוכנית מעבירה לפונקציה ctime את כתובת הפרמטר system_time תוך שימוש בטכניקה המתוארת בפרק 10. בתחילת התוכנית נעזרים ב- include כדי להעתיק לתוכנית את קובץ הכותר time.h המאפשר שימוש בפונקציות הספריה הדרושות.

בצורה דומה, בתוכנית **SQRT.CPP** מתואר השימוש בפונקציה sqrt לחישוב שורש ריבועי של מספר ערכים המוצגים על המסך. הצהרת (אב-טיפוס) פונקציית הספריה sqrt נמצאת בקובץ הכותר math.h.

```
#include <iostream.h>
#include <math.h>              // Contains sqrt prototype

void main(void)
 {
 cout << "The square root of 100.0 is " << sqrt(100.0) << endl;
 cout << "The square root of 10.0 is " << sqrt(10.0) << endl;
 cout << "The square root of 5. 0 is " << sqrt(5.0) << endl;
 }
```

ולבסוף, בתוכנית **SYSCALL.CPP** אנו משתמשים ב**פונקציית מערכת** (System function). אב הטיפוס של הפונקציה מוגדר ב**קובץ stdlib.h**. פונקציית המערכת מאפשרת לתוכניות להריץ בקלות פקודות של מערכת ההפעלה (למשל, dir של DOS), או תוכניות אחרות:

```
#include <iostream.h>
#include <stdlib.h>

void main(void)
  {
    system ("DIR");
  }
```

בתוכנית זו משמשת פונקציית המערכת להרצת הפקודה DIR של מערכת הפעלה DOS. כדאי לנסות ולהריץ באופן דומה פקודות DOS אחרות, או תוכניות אשר נכתבו בפרקים הקודמים.

את התוכנית ניתן להריץ באמצעות קובץ exe בלבד. על כן, תוכנית זו לא תפעל באמצעות מהדר התרגול Tclite. לשם הרצה צריך להשתמש במהדר בגירסה מלאה.

כיצד להבין את פונקציות הספריה

בדרך כלל, המהדרים של שפת ++C מכילים מספר רב של פונקציות ספריה ולרוב, בתיעוד המצורף למהדר יש תיאור מפורט של פונקציות אלו. בדרך כלל מוצגות ההצהרות (אב-טיפוס) של הפונקציות. לדוגמה, הצהרת הפונקציה sqrt:

```
double sqrt(double);
```

במקרה זה, אנו לומדים מהצהרת הפונקציה שהערך המוחזר שלה הוא מטיפוס double, כאשר גם הפרמטר שהיא מצפה לקבל הוא מאותו טיפוס.

בדומה לפונקציה זו, הצהרת פונקציית הספריה time היא:

```
time_t time(time_t *);
```

בדוגמה זו אנו רואים שהפונקציה מורה שהפונקציה מחזירה ערך מהטיפוס time_t (המוגדר בקובץ כותר time.h). הפונקציה מצפה לפרמטר מצביע למשתנה מסוג time_t.

מומלץ להתבונן היטב בתיעוד על פונקציות הספריה השונות המסופקות עם המהדרים של שפת ++C, כדי להימנע משגיאות בעת הקריאה לפונקציות.

ראוי לציין כאן שהמתכנת יכול לצרף את הפונקציות שהוא כותב אל הספריות הסטנדרטיות המסופקות על ידי יצרן המהדר. כך הוא יכול לכלול בהן פונקציות שהוא ערך בעצמו ושדרושות לו בתוכניות שהוא מפתח. כפי שמתכנתים שונים יכולים להשתמש בספריות ההרצה הסטנדרטיות, כך הם יכולים להשתמש בספריות שאחד מחבריהם הכין, או לרכוש ספריות מוכנות כאלה (יש גם Shareware).

פונקציות API

מהדרים רבים תומכים בפונקציות ממשק תכנות יישומים - API (**A**pplication **P**rogram **I**nterface), בנוסף לספריית זמן הריצה (Run-time library) הסטנדרטית של ++C. בסביבת Windows למשל, קיימות פונקציות API שונות: גרפיקה, תקשורת, מולטימדיה ורבות אחרות. לכן, לפני שכותבים פונקציה שעשויה למלא משימה סטנדרטית, כדאי לברר מהן פונקציות API, שכבר כלולות בספריית המהדר.

סיכום

ספריות ההרצה של הפונקציות בשפת ++C מכילות אוסף מקיף של פונקציות מוכנות שניתנות לשימוש בכל תוכנית ותוכנית. על ידי ניצול יתרונותיה של ספריית ההרצה ניתן לחסוך זמן רב בפיתוח תוכניות. מומלץ ללמוד היטב את מבנה הפונקציות על פי התיעוד הנלווה למהדר. לפני שנעבור לפרק הבא מומלץ לבחון אם מובנים הנושאים הבאים:

- ✓ ספריות ההרצה של הפונקציות הן אוסף של פונקציות מוכנות לשימוש בתוכניות השונות.
- ✓ כדי להשתמש בפונקציית ספריה יש להצהיר עליה כתוכנית.
- ✓ לרוב המהדרים של שפת ++C מצורפים קבצי כותר המכילים הצהרות תקינות של הפונקציות שבספריית ההרצה שלהם.

בפרק 12 נציג את המשתנים המקומיים וטווח הכרתם. טווח הכרת המשתנה הוא הטווח בתוכנית שבו ניתן להשתמש במשתנה מסוים.

תרגילים

1. כיתבו פונקציה המקבלת שלושה ערכים מטיפוס float, ומדפיסה את הערך log של הראשון, ערך sin של השני ואת ערך cos של השלישי. היעזרו בספריה המתמטית של ++C.

2. כיתבו תוכנית המדפיסה את השעה הנוכחית, סופרת בלולאה מ-1 עד 1,000,000 ומדפיסה שוב את השעה הנוכחית.

פרק 12

משתנים מקומיים וטווח הכרתם

בפרקים הקודמים למדנו שהפונקציות הן יחידות תכנות קטנות המרכיבות את התוכנית. הפונקציות שהצגנו בפרקים הקודמים מבצעות פעולות פשוטות. כאשר כותבים פונקציות מורכבות יותר, מתעורר צורך להגדיר משתנים בתוך הפונקציות כדי שהן תוכלנה לבצע את המשימות המוגדרות להן. המשתנים המוגדרים בתוך הפונקציה נקראים **משתנים מקומיים** (**Local variables**). ערך המשתנה המקומי ועצם קיומו מוכר אך ורק לפונקציה שבה הוא מוגדר. לדוגמה, משתנה salary המוגדר בפונקציה payroll מוכר אך ורק בפונקציה זו. לשאר הפונקציות אין גישה לערך המאוחסן בו.

בהמשך הפרק נדון בנושאים הבאים:

- הגדרת **משתנים מקומיים** (Local variables) בתוך פונקציה, נעשית על ידי ציון השם והטיפוס שלהם, כמו שהדבר נעשה בפונקציה main.
- שמות המשתנים בתוך פונקציה צריכים להיות ייחודיים, אך פונקציות שונות יכולות להשתמש באותם שמות משתנים.
- **טווח ההכרה** (Scope) של משתנה הוא אוסף המקומות בתוכנית בהם המשתנה מוכר ונגיש.
- **משתנים גלובליים** (Global variables) מוכרים לכל אורך התוכנית, ונגישים בכל פונקציה.
- **אופרטור טווח ההכרה** (**::**, Global resolution operator) מסייע בבקרת טווח ההכרה של משתנה.

תהליך הגדרת משתנים מקומיים קל מאוד, ולמעשה כבר הגדרנו משתנים מסוג זה בפונקציות main בתוכניות המוצגות בפרקים הקודמים.

הגדרת משתנה מקומי

משתנה מקומי (**Local variable**) הוא משתנה המוגדר בתוך הפונקציה. המשמעות של המילה **מקומי** היא שהמשתנה מוכר אך ורק בתחום הפונקציה שבה הוא מוגדר. משתנים מקומיים מוגדרים בפונקציה מייד בתחילת קוד ההגדרה של הפונקציה.

```
void some_function(void)
 {
   int count;
   float result;
 }
```

בתוכנית **USEBEEPS.CPP** תפקיד הפונקציה sound_speaker הוא לגרום שברמקול המחשב יושמע מספר צפצופים על פי הערך המועבר בפרמטר beeps. בפונקציית sound_speaker מוגדר משתנה מקומי counter המשמש כמשתנה עזר בלולאה של השמעת הצפצופים.

```
#include <iostream.h>

void sound_speaker(int beeps)
 {
   int counter;

   for (counter = 1; counter <= beeps; counter++)
     cout << '\a';
 }

void main(void)
 {
   sound_beeps(2);
   sound_beeps(3);
 }
```

יש להדגיש, כי את המשתנה המקומי counter מגדירים מייד בתחילת הקוד של הפונקציה.

בעיית התנגשות של שמות משתנים

במהלך התכנות של פונקציות אנו עלולים להגדיר שמות זהים למשתנים מקומיים בפונקציות שונות. כפי שציינו קודם, משתנה מקומי מוכר אך ורק בתחום הפונקציה שבה הוא מוגדר. שפת C++ דואגת לייחס כל משתנה לפונקציה הנכונה. על כן אין התנגשות של שמות המשתנים, על אף שהם זהים. בתוכנית **LCLNAME.CPP** תפקיד הפונקציה add_values הוא לחבר שני מספרים שלמים. בפונקציה מוגדר משתנה מקומי value המאחסן בתוכו את תוצאת חיבור המספרים. אולם גם בפונקציה main

מוגדר משתנה מקומי בשם value, המשמש פרמטר להעברת מידע לפונקציה add_values. שמות זהים אלה אינם מתנגשים ביניהם, מכיון ששפת ++C מייחסת כל משתנה מקומי לפונקציה שבה הוא מוגדר.

```
#include <iostream.h>

int add_values(int a, int b)
 {
   int value;

   value = a + b;

   return(value);
 }

void main(void)
 {
   int value = 1001;
   int other_value = 2002;

   cout << value << " + " << other_value << " = " <<
               add_values(value, other_value) << endl;
 }
```

המשתנה המקומי

משתנה מקומי הוא משתנה המוגדר **בתוך** הפונקציה. שם המשתנה והערך המאוחסן בו מוכרים אך ורק בתחום הפונקציה שבה הוא מוגדר. משתנים מקומיים מוגדרים בפונקציה מייד בתחילתה.

השמות שניתנים למשתנים מקומיים צריכים להיות ייחודיים רק בתחום הפונקציה שבה הם מוגדרים (ועל כך אפשר להגדיר משתנה בשם זהה בכל פונקציה אחרת). כאשר מגדירים משתנה מקומי בתוך פונקציה, אפשר לקבוע לו ערך התחלתי בעזרת אופרטור ההשמה (Assignment operator).

הגדרת משתנה גלובלי

משתנים מקומיים, כך ראינו, מוגדרים ומוכרים רק בתוך הפונקציות שבהן הם מוגדרים. לעומתם, בשפת ++C קיים סוג משתנה המוכר **לכל** הפונקציות בתוכנית. משתנה מסוג זה נקרא **משתנה גלובלי** (**Global variable**), כי הוא כללי לכל הפונקציות. משתנים גלובליים מוגדרים בתחילת התוכנית, ומחוץ להגדרות הפונקציות שבתוכנית.

```
int some_global_variable;
```
הגדרת משתנה גלובלי

```
void main(void)
 {
   // Program statements would be here
 }
```

בתוכנית **GLOBAL.CPP** מוגדר משתנה גלובלי number לכל הפונקציות בתוכנית גישה אליו, ועל כן הן מסוגלות להשתמש ולשנות את הערך המאוחסן בו. בתוכנית זו, כל פונקציה מציגה את ערך המשתנה ומקדמת את ערכו באחד.

```
#include <iostream.h>

int number = 1001;

void first_change(void)
 {
  cout << "number's value in first_change " << number << endl;
  number++;
 }

void second_change(void)
 {
  cout << "number's value in second_change " << number <<endl;
  number++;
 }

void main(void)
 {
  cout << "number's value in main " << number << endl;
  number++;
  first_change();
  second_change();
 }
```

ככלל, יש להימנע משימוש במשתנים גלובליים בתוכניות ++C. מכיון שלכל פונקציה בתוכנית יש גישה אליהם, קשה לעקוב ולדעת באיזה שלב בתוכניות ועל ידי איזו פונקציה נעשה שינוי בערכם. לכן, עדיף ליישם את דרך התכנות שבה מוגדרים משתנים מקומיים בפונקציה main, אשר משמשים פרמטרים להעברת ערכים לפונקציות הספציפיות הדורשות משתנים אלה. יש לציין ולהזכיר כי בהעברת מידע דרך פרמטרים, ערך המשתנים המועברים מועתק למחסנית, כך שהפונקציות המקבלות את המידע משתמשות במחסנית, ולא פוגעות בערכים המקוריים.

בעיית משתנה גלובלי ומשתנה מקומי בעלי שם זהה

יש להימנע משימוש במשתנים גלובליים. במקרה בו שם משתנה גלובלי ושם משתנה מקומי זהים יש התנגשות. הנה דרך פתרון לבעיה: אופרטור טווח ההכרה (**::**) של שפת C++ פותר את הבעיה ומאפשר להתייחס למשתנה הגלובלי בתוך פונקציה שבה מוגדר משתנה מקומי באותו שם. למשל, נניח שבתוכנית מוגדרים שני משתנים בשם number, אחד גלובלי ואחד מקומי. במשפט הבא (מתוך הפונקציה) התוכנית מתייחסת למשתנה המקומי.

```
number = 1001;     // Local variable reference
```

לעומת זאת, במשפט הבא משתמשים באופרטור טווח ההכרה כדי להתייחס למשתנה הגלובלי בתוך הפונקציה שבה מוגדר משתנה מקומי באותו שם.

```
::number = 2002;         // Global variable reference
```

בתוכנית **GLOBLOCA.CPP** מוגדר משתנה גלובלי בשם number. בנוסף, בפונקציה show_numbers מוגדר משתנה מקומי, גם הוא בשם number. בתוך הפונקציה מופעל האופרטור טווח ההכרה כדי לאפשר גישה למשתנה הגלובלי.

```
#include <iostream.h>

int number = 1001;       // Global variable

void show_numbers(int number)
 {
  cout << "Local variable number contains " << number << endl;

  cout << "Global variable number contains " << ::number << endl;
 }

void main(void)
 {
   int some_value = 2002;

   show_numbers(some_value);
 }
```

לאחר הידור והרצת התוכנית יוצגו על המסך שורות הפלט הבאות:

```
C:\> GLOBLOCA  <Enter>
Local variable number contains 2002
Global variable number contains 1001
```

על פי התוצאות ניתן להיווכח שהשימוש באופרטור טווח ההכרה מאפשר גישה למשתנים גלובלים בתוך הפונקציה שבה מוגדר משתנה מקומי בשם זהה. יחד עם זאת, השימוש במשתנים גלובלים ומקומיים בעלי שמות זהים עלול לגרום לבלבול שתוצאותיו מובילות לתקלות ושגיאות מיותרות. על כן מומלץ להמעיט בשימוש במשתנים גלובליים בתוכנית.

המשתנה הגלובלי

משתנה גלובלי הוא משתנה המוכר לכל הפונקציות בתוכנית. משתנה גלובלי מוגדר בתחילת קובץ מקור, ומחוץ להגדרות הפונקציות שבתוכנית. לכל הפונקציות שנמצאות מתחת להגדרת המשתנים הגלובלים יש גישה למשתנים, ולכן הן מסוגלות להשתמש בהם ולשנות את ערכם. מומלץ להמעיט בשימוש במשתנים גלובליים עקב בעיות ותקלות שעלולות להיות בהגדרת שמות משתנים אלה, ובגישה החופשית אליהם מכל מקום בתוכנית.

טווח ההכרה - Variable's Scope

בספרות המקצועית הקשורה לשפת ++C מרבים להתייחס למונח טווח (Scope), המגדיר את חלקי התוכנית בהם שם המשתנה מוכר (ולכן פעיל). טווח ההכרה של משתנה מקומי הוא בפונקציה בה הוא מוגדר בלבד. לעומת זאת, טווח ההכרה של משתנה גלובלי הוא כל התוכנית, ולכן טווח ההכרה שלו רחב יותר מטווח ההכרה של משתנה מקומי.

סיכום

בפרק זה למדנו להגדיר משתנים מקומיים המוכרים בתוך הפונקציה שבה הם מוגדרים בלבד. סוג משתנה אחר שהוצג בפרק הוא המשתנה הגלובלי המוכר לאורך **כל** התוכנית. מכיון שהמשתנים הגלובליים יכולים לגרום לשגיאות שקשה לגלותן, עדיף להימנע מהשימוש בהם ככל האפשר.

בפרק 13 נלמד כיצד ++C מאפשרת לתוכניות להגדיר שתי פונקציות או יותר, אשר נושאות את אותו השם, אך מתייחסות לפרמטרים שונים ואף לטיפוס החזרה (Return) שונה. על ידי העמסת שם הפונקציה בדרך זו אפשר לפשט מאוד את השימוש בפונקציות.

לפני שנעבור לפרק הבא, מומלץ לבחון אם מובנים הנושאים הבאים:

- ✓ משתנים מקומיים הם משתנים המוגדרים בתוך הפונקציה.
- ✓ המשתנים המקומיים מוכרים רק בתוך הפונקציות שבהן הם מוגדרים.
- ✓ בפונקציות שונות ניתן להגדיר משתנים מקומיים בעלי שמות זהים, ללא התנגשות.

- ✓ משתנים גלובליים מוכרים לאורך כל התוכנית.
- ✓ כדי ליצור משתנה גלובלי יש להגדיר אותו בתחילת קובץ המקור ומחוץ לכל הפונקציות.
- ✓ שימוש במשתנים גלובליים יכול לגרום לתקלות שקשה לזהות. מפני שהערך של משתנים הגלובליים מושפע מפעולות בכל התוכנית. לכן מומלץ להימנע עד כמה שאפשר משימוש במשתנים גלובליים.

תרגילים

1. מהו הפלט של התוכנית הבאה?

```
#include <iostream.h>

int b = 5;
void f (int a)
  {
    int b = a;
    b = b + 3;
    cout << a << endl;
    cout << b << endl;
    cout << ::b << endl;
    cout << "=======\n";
  }

void main(void)
  {
    int a = 7;
    f(a);
    f(17);
    f(5);
    b = 100;
    f(a+2);
  }
```

2. מהו הפלט של התוכנית הבאה?

```
#include<iostream.h>

int x = 9;
int f (int x)
  {
   int tmp = 1;    return x + tmp;
  }
int g (int x)
  {
   int tmp = 1;    return ::x - tmp;
  }
void main(void)
  {
    cout << f(x) << endl;   cout << g(x) << endl;
    cout << f(g(x)) << endl;

    int x = 3;
    cout << g(x) << endl; cout << f(x) << endl;
    cout << f(g(x)) << endl;
    cout << f(g(::x)) << endl; cout << g(f(x)) << endl;
    cout << g(3) << endl;

    x = 111;
    cout << g(7) << endl;    cout << f(7) << endl;

    ::x = 111;
    cout << g(7) << endl;    cout << f(7) << endl;
  }
```

פרק 13

העמסת פונקציות

בפרקים הקודמים למדנו שבהגדרת פונקציה יש לציין את טיפוס הערך המוחזר על ידה, שמות הפרמטרים שהפונקציה מנהלת, והטיפוס של כל אחד מהם. לפיכך, כאשר נרצה לכתוב פונקציה לחיבור שני מספרים ונקרא לה בשם add_values ואחר כך נרצה לחבר שלושה מספרים, עלינו להגדיר פונקציה חדשה דומה לקודמת ולתת לשתיהן שמות שונים כדי לזהות אותן באופן מוחלט: לדוגמה, בפונקציה, add_two_values ובפונקציה add_three_values. כדי למנוע כפילויות אלו, שפת ++C מצוידת במנגנון המאפשר להגדיר מספר פונקציות בעלות שם זהה. בזמן ההידור, המהדר בוחן את מספר וטיפוס הפרמטרים בכל קריאה לפונקציה ומציב לכל קריאה וקריאה את הפונקציה הנכונה. תהליך זה נקרא **העמסת יתר של פונקציות** (**Overloading**), או בקיצור **העמסת פונקציות**.

בהמשך הפרק נדון בנושאים הבאים:

- מושג העמסת פונקציות והמקרים בהם רצוי ליישם אותו.
- העמסת פונקציות בתוכנית.

העמסת פונקציות היא תכונה של שפת ++C, שאינה קיימת בשפת C. כפי שנראה בהמשך, ניצול העמסת פונקציות תורם לבהירות והבנת קוד התוכנית.

מהי העמסת פונקציות

מנגנון העמסת פונקציות מאפשר להגדיר מספר פונקציות שונות שבהם זהים השם וטיפוס הערך המוחזר. לדוגמה, בתוכנית הבאה מנצלים את המנגנון להעמסת הפונקציה add_values. הפונקציה הראשונה מבצעת חיבור של שני ערכים מטיפוס int. לעומתה, ההגדרה השנייה של הפונקציה מבצעת חיבור של שלושה ערכים. במהלך ההידור, שפת ++C קובעת לכל קריאה את ההגדרה המתאימה.

```
#include <iostream.h>

int add_values(int a, int b)
 {
   return(a + b);
 }

int add_values(int a, int b, int c)
 {
   return(a + b + c);
 }

void main(void)
 {
   cout << "200 + 801 = " << add_values(200, 801) << endl;
   cout << "100 + 201 + 700 = " << add_values(100, 201,
                                          700) << endl;
 }
```

בתוכנית זו ניתן לראות שהפונקציה add_values מוגדרת פעמיים. בהגדרה הראשונה היא מחברת שני ערכים מטיפוס int, ובהגדרה השנייה היא מחברת שלושה ערכים. אין צורך לציין בתוכנית שהפונקציות מועמסות. המהדר של שפת ++C מבדיל בין ההגדרות השונות לפי השוני בפרמטרים שמועברים לפונקציה.

בצורה דומה, בתוכנית **MSG_OVR.CPP** קיימת העמסת הפונקציה show_message. בהגדרה הראשונה מציגים על המסך הודעת פלט כאשר אין מעבירים לפונקציה פרמטרים כלשהם. בהגדרה השנייה מציגים על המסך הודעת פלט אחת, המועברת כפרמטר לפונקציה. בהגדרה השלישית והאחרונה מציגים על המסך שתי הודעות פלט המועברות כפרמטרים לפונקציה.

```
void show_message(void)
 {
   cout << "Default message: Rescued by C++" << endl;
 }
void show_message(char *message)
 {
   cout << message << endl;
 }

void show_message(char *first, char *second)
 {
   cout << first << endl;
   cout << second << endl;
 }
void main(void)
 {
   show_message();
   show_message("I've Been Rescued!");
   show_message("C++ is not so hard!", "Overloading is cool!");
 }
```

מתי להשתמש בהעמסת פונקציות?

המקרה הנפוץ ביותר לשימוש בהעמסת פונקציות הוא כאשר משתמשים בפונקציה לביצוע מטלה, למרות שטיפוסי הפרמטרים המוזנים לה יכולים להיות שונים מדי פעם. למשל, נניח כי בתוכנית קיימת פונקציה בשם day_of_week המחזירה מספר המתאים ליום בשבוע (0 ליום ראשון, 1 ליום שני, ... ו- 6 ליום שבת). אז ניתן לבצע העמסה של פונקציה זו, כדי שתתן את התוצאה הנכונה, בין אם הקלט הוא תאריך ג'וליאני (Julian day) ובין אם הקלט מורכב משלושה פרמטרים המציגים את היום, החודש והשנה של תאריך מסוים.

```
int day_of_week(int julian_day)
 {
   // Statements
 }

int day_of_week(int month, int day, int year)
 {
   // Statements
 }
```

בפרקים הבאים נעסוק בתכונות של שפת C++ כשפה לתכנות מונחה עצמים. העמסת פונקציות היא חלק מתכונות אלו המעניקות עוצמה ליכולת הביצוע של תוכניות מחשב.

העמסת פונקציות משפרת את קריאות התוכנית

העמסת פונקציות (Function overloading) מאפשרת להגדיר בתוכנית אחת מספר פונקציות בעלות שם זהה. הפונקציות המועמסות צריכות להחזיר ערך מאותו הטיפוס, אך הן נבדלות זו מזו במספר, או בטיפוסי הפרמטרים שהן מקבלות. בעבר היה על מתכנתי C ליצור מספר פונקציות בעלות שמות דומים, וגם לזכור מהם הפרמטרים אשר מתאימים לכל אחת מהן. כאשר משתמשים בהעמסת פונקציות, לעומת זאת, על המתכנת לזכור שם פונקציה אחד בלבד.

סיכום

מנגנון העמסת פונקציות מאפשר להגדיר מספר פונקציות בעלות שם זהה. במהלך ההידור שפת ++C קובעת לכל קריאה את ההגדרה המתאימה, לפי המספר והטיפוס של הפרמטרים המוגדרים ומועברים לפונקציה.

לפני שנעבור לפרק הבא מומלץ לבחון אם מובנים הנושאים הבאים:

- ✓ העמסת פונקציות מאפשר ליצור גרסאות שונות לאותה פונקציה.
- ✓ כדי להעמיס פונקציה, צריך להגדיר שתי פונקציות, או יותר, באותו שם ועם טיפוס החזרה. השוני בין הפונקציות הוא רק במספר ובטיפוס של הפרמטרים שהן מקבלות.
- ✓ בזמן ההידור יקבע המהדר איזו פונקציה תופעל בעת הצורך, לפי המספר וטיפוס הפרמטרים המוגדרים והמועברים לה.
- ✓ העמסת פונקציות מפשטת את תהליכי התכנות, בכך שהן מאפשרות למתכנתים לפעול עם אותו שם פונקציה, כאשר התוכניות שלהם צריכות למלא משימה מוגדרת.

בפרק הבא נציג כיצד ניתן להגדיר משתנה מיוחס בשפת ++C. פרמטרים אלה מקלים על הניהול והטיפול בפרמטרים המועברים לפונקציה.

תרגילים

1. כיתבו את הפונקציות min ו-max (ראו תרגיל 3 בפרק 9) עבור ארגומנטים מטיפוס long וארגומנטים מטיפוס float. בידקו את הפונקציות האלו על ידי הרצת דוגמאות.
2. כיתבו פונקציות המחשבות את הממוצע של 2, 3 ו-4 ארגומנטים (מטיפוס float).
3. כיתבו תוכנית שמקבלת ערך n מהמשתמש ובודקת אותו: אם n>3 התוכנית מדפיסה n כוכביות ('*'). אחרת, התוכנית מדפיסה 5 סימני דולר ('$').

פרק 14

המשתנה המיוחס בשפת ++C

בפרק 10 הצגנו כיצד ניתן לשנות את ערכם של הפרמטרים על ידי שימוש במצביעים למשתנים. גם הראינו, כי כדי להצביע על משתנה צריך להציב כוכבית לפני שם הפרמטר בפונקציה. שיטה זו מקורה בצורת התכנות בשפת C. כדי להקל ולצמצם את הצעדים הדרושים לשינוי ערך של פרמטר בשפת ++C, כוללת השפה את המושג **יחס** (**Reference**). בפרק הזה, נראה שיחס או **משתנה מיוחס** הוא למעשה כינוי (שם חליפי) למשתנה המוגדר בזיכרון.

בהמשך הפרק נדון בנושאים הבאים:

- הצהרת משתנה מיוחס ואתחולו נעשית על ידי כתיבת הסימן & מייד לאחר טיפוס המשתנה ושימוש באופרטור ההשמה כדי להקצות לו ערך. לדוגמה:

```
int& alias_name = variable;
```

- כאשר מעבירים לפונקציה פרמטר לפי ייחוס, היא יכולה לשנות את ערכו מבלי להשתמש במצביע.
- בכותרת הפונקציה מצהירים על פרמטר ייחוס על-ידי כתיבת הסימן & לאחר טיפוסו. ניתן אז לשנות את ערכו של הפרמטר בתוך פונקציה, מבלי להשתמש במצביע.

השימוש ביחס הופך את שינוי הפרמטרים לפעולה פשוטה מאוד. נכנה אותה: שינוי פרמטרים לפי ייחוס.

משתנה מיוחס

משתנה מיוחס (Reference) מאפשר לציין שם חליפי, או כינוי, למשתנה המוגדר בתוכנית. הגדרת משתנה מיוחס מתבצעת על ידי הצבת הסימן & מייד לאחר כתיבת טיפוס המשתנה לפי הדוגמה הבאה:

הגדרת משתנה מיוחס

```
int& alias_name = variable;
```

לאחר הגדרת המשתנה המיוחס ניתן להשתמש בו כמו במשתנה רגיל. הנה כך:

```
alias_name = 1001;
variable = 1001;
```

בתוכנית **SHOW_REF.CPP** מוגדר משתנה מיוחס alias_name המשמש כינוי למשתנה number. בתוכנית משתמשים במשתנה הרגיל ובמשתנה המיוחס, באופן חלופי.

```
#include <iostream.h>

void main(void)
 {
  int number = 501;
  int& alias_name = number;     //Create the reference

  cout << "The variable number contains " << number << endl;
  cout << "The alias to number contains " << alias_name << endl;
  alias_name = alias_name + 500;

  cout << "The variable number contains " << number << endl;
  cout << "The alias to number contains " << alias_name << endl;
 }
```

במהלך התוכנית מוסיפים את הערך 500 לערך הקיים במשתנה המיוחס alias_name. כתוצאה מפעולה זו, ערך זה מתווסף למשתנה number. לאחר הידור התוכנית והרצתה יוצגו על המסך שורות הפלט הבאות:

```
C:\> SHOW_REF  <Enter>
The variable number contains 501
The alias to number contains 501
The variable number contains 1001
The alias to number contains 1001
```

השימוש במשתנה מיוחס במהלך התוכנית אינו מומלץ, מכיון שהדבר מקשה על הבנת התוכנית. לעומת זאת, השימוש במשתנה מיוחס מקל מאוד על שינוי ערכי הפרמטרים בפונקציה, כפי שנתאר בהמשך.

הגדרת משתנה מיוחס בתוכנית

המשתנה המיוחס הוא שם חליפי או כינוי למשתנה המוגדר בתוכנית. כדי להגדיר משתנה מיוחס כותבים את הסימן "&" מייד לאחר טיפוס המשתנה. ולאחר ציון שם המשתנה המיוחס כותבים את הסימן שווה "=" שבהמשכו נמצא שם המשתנה לו מתייחסים, לדוגמה:

```
float& salary_alias = salary;
```

העברת פרמטרים לפי ייחוס

המטרה העיקרית של משתני ייחוס היא ליישם **העברת פרמטרים לפי ייחוס**, ועל ידי כך להקל על שינוי ערכי הפרמטרים שבתוך הפונקציה. בתוכנית הבאה **REFERNC.CPP**, מוגדר משתנה מיוחס בשם number_alias המשמש כינוי למשתנה number. המשתנה המיוחס משמש כפרמטר המועבר לפונקציה change_value. בפונקציה זו מציבים לפרמטר את הערך 1001.

```
#include <iostream.h>

void change_value(int &alias)
 {
  alias = 1001;
 }

void main(void)
 {
  int number;
  int& number_alias = number;

  change_value(number_alias);

  cout << "The variable number contains" << number << endl;
 }
```

בתוכנית מועבר המשתנה המיוחס לפונקציה change_value כפרמטר. יש לציין שבהגדרת הפונקציה change_value מוגדר הפרמטר כמשתנה מיוחס מטיפוס int.

```
void change_value(int& alias)
```

בדרך זו ניתן לשנות את ערך הפרמטר שבפונקציה ללא צורך במצביעים. הימנעות משימוש בכוכבית (*) בתוך הפונקציה מקל על כתיבתה והבנתה. שיטת העברת פרמטרים זו נקראת **העברה לפי ייחוס** (**By reference**).

שימוש בהערות לציון שימוש בהעברה לפי ייחוס

רוב מתכנתי ++C קשורים לטכניקות תכנות של שפת C, ובכללן השימוש במצביעים, כשיטה לשינוי ערכם של משתנים בתוך הפונקציות. עובדה זו גורמת למתכנתים לחשוב שבהעדר מצביעים, אין ערך הפרמטר משתנה בפונקציה. כדי למנוע אי הבנות מסוג זה מומלץ להרבות ברישום הערות בתוכניות שבהן משתמשים בטכניקת העברת פרמטרים לפי ייחוס. בדרך זו, המתכנתים בשפת ++C יבינו טוב יותר את ביצועי הפונקציות בתוכנית.

דוגמה נוספת

הפונקציה הבאה מוצגת בפרק 10, ותפקידה להחליף בין ערכי שני פרמטרים מטיפוס float:

```
void swap_values(float *a, float *b)
 {
   float temp;
   temp = *a;
   *a = *b;
   *b = temp;
 }
```

בפונקציה זו יש שימוש מעורב של משתנים שהם מצביעים ושל משתנים שאינם מצביעים. בתוכנית הבאה, **SWAP_REF.CPP**, מנצלים את תכונותיהם של המשתנים המיוחסים ועל ידי כך מצמצמים את מורכבות הגדרת הפונקציה.

```
#include <iostream.h>

void swap_values(float& a, float& b)
 {
   float temp;
   temp = a;
   a = b;
   b = temp;
 }

void main(void)
 {
   float big = 10000.0;
   float small = 0.00001;
   float& big_alias = big;
   float& small_alias = small;
   swap_values(big_alias, small_alias);
   // swap_values(big, small);
   cout << "Big contains " << big << endl;
   cout << "Small contains " << small << endl;
 }
```

ניתן לראות שהגדרת הפונקציה swap_values מובנת יותר מגרסתה הקודמת. לעומת זאת, בתוכנית נוספו שני שמות הפרמטרים המיוחסים big_alias ו-small_alias, הדרושים ליישום השיטה.

כללי העברת פרמטרים לפי ייחוס

המשתנה המיוחס אינו משתנה רגיל. כאשר מייחסים ערך למשתנה מיוחס, הערך המאוחסן בו לא ניתן לשינוי. להבדיל ממשתנה מצביע, קיימות מספר פעולות שאינן ניתנות לביצוע במשתנה מיוחס:

- לא ניתן לקבל את הכתובת של המשתנה המיוחס על ידי שימוש באופרטור "כתובת של...." (&).
- לא ניתן לשייך מצביע למשתנה מיוחס.
- לא ניתן לבצע השוואות לוגיות של משתנים מיוחסים.
- לא ניתן לבצע פעולות אריתמטיות במשתנים מיוחסים.
- אין אפשרות לשנות את שיוך המשתנה המיוחס לאחר קביעתו.

בפרקים הבאים, כאשר נציג את התכונות של תכנות מונחה עצמים בשפת ++C, נחזור ונעסוק ביתר פירוט במשתנה המיוחס.

שימוש במשתני ייחוס לשינוי פרמטרים של פונקציות

כזכור לנו מפרק 10, תוכנית יכולה לשנות ערך של פרמטרים בתוך פונקציות על ידי שימוש במצביעים. פונקציה יכולה לשנות ערך של פרמטר רק אם היא מקבלת את כתובתו. כדי לקבל את כתובת הפרמטר, משתמשים באופרטור הכתובת (&). הפונקציה משתמשת במשתנים מצביעים (אשר מאחסנים כתובות זיכרון). כדי להצהיר על משתנה מצביע בתוך פונקציה, צריך לכתוב כוכבית (*) לפני שמו. כדי להשתמש בערך של הפרמטר בתוך הפונקציה, או כדי לשנותו, מוסיפים לפני כל התייחסות אליו את אופרטור ההצבעה (*). לרוע המזל, פעולות רבות בתוך פונקציות משלבות שימוש במשתנים מצביעים ושאינם מצביעים.

שילוב סוגים שונים של מצביעים עלול לגרום לבלבול ולתקלות. משתני ייחוס של ++C מפשטים את תהליך שינוי ערכם של פרמטרים, על ידי עקיפה של הצורך במשפטים המשלבים שימוש במשתנים מצביעים ושאינם מצביעים. המחיר של נוחות זו הינו תוספת משתנים, שיש צורך להכיר בעת קריאת התוכנית.

סיכום

בפרק זה למדנו כיצד להעביר פרמטרים לפונקציה לפי ייחוס. הצגנו את המשתנה המיוחס כשם חליפי, או כינוי, למשתנה שבזיכרון של המחשב. על ידי העברת פרמטרים לפי ייחוס ניתן לשנות את ערכי הפרמטרים בתוך הפונקציה. לפני שנעבור לפרק הבא נבחן אם מובנים הנושאים הבאים:

- ✓ המשתנה המיוחס הוא שם חליפי, או כינוי, למשתנה בתוכנית.
- ✓ כדי להגדיר משתנה מיוחס מציבים את הסימן "&" מייד לאחר טיפוס המשתנה.
- ✓ לאחר שיוך משתנה למשתנה המיוחס, ערכו של המשתנה המיוחס לא ניתן לשינוי.
- ✓ מומלץ להרבות ברישום הערות בתוכנית כדי לציין את השימוש בטכניקת העברת פרמטרים לפי ייחוס. זאת, כדי להגביר את קריאות התוכנית ולהזכיר שבפונקציות אלו ערכם של הפרמטרים עשויים להשתנות.
- ✓ שימוש מורחב של המשתנים המיוחסים בתוך התוכנית עלול לגרום לסיבוך קוד התוכנית ולפגוע בהבנת התהליכים המתבצעים בה.

בפרק הבא נציג כיצד ניתן להגדיר ערכי ברירת המחדל לפרמטרים בפונקציה.

תרגילים

1. מהו הפלט של התוכנית הבאה?

```
#include <iostream.h>

void main(void)
  {
    long  y = 713;
    long& x = y;
    long  z = y;
    cout << y << " " << x << " " << z << endl;

    x = 313;
    cout << y << " " << x << " " << z << endl;

    y = 500;
    cout << y << " " << x << " " << z << endl;

    z = 414;
    cout << y << " " << x << " " << z << endl;
  }
```

2. כיתבו תוכנית שבה כלולות שלוש הפונקציות הבאות:

- **dec** - מקבלת משתנה value (Reference variable) וערך delta, ומורידה מ-value את delta. כל הטיפוסים הם טיפוסי נקודה צפה (Float).
- **inc** - כמו בפונקציה DEC, אבל מוסיפה את delta ל-value.
- **print** - מדפיסה את ערך הארגומנט שלה.

3. כיתבו את הפונקציה swap להחלפת ערכי משתנים (מסוג float) ושאינה משתמשת במשתנה עזר נוסף.

פרק 15

קביעת ערכי ברירת-מחדל לפרמטרים

בפרקים הקודמים למדנו שבשפת ++C ניתן להעביר ערכים לפונקציות תוך שימוש בפרמטרים שלה. בפרק 13 הצגנו את מנגנון העמסת פונקציות, שבעזרתו ניתן להגדיר מספר פונקציות המשתמשות באותו שם. הפונקציות נבדלות ביניהן על ידי המספר וטיפוס הפרמטרים. תכונה נוספת של שפת ++C מאפשרת לקרוא לפונקציה ללא ציון הפרמטרים. במקרה זה, השפה תדאג להציב לפרמטרים אלו ערכי ברירת מחדל המוגדרים מראש.

בפרק זה נדון בנושאים הבאים:

- ❖ שפת ++C מאפשרת להגדיר ערכי ברירת-מחדל לפרמטרים של פונקציה.
- ❖ ערכי ברירת המחדל מופיעים בכותר הפונקציה (Function header) בעת הגדרתה.
- ❖ כאשר בקריאה לפונקציה מושמטים אחד או יותר מערכי הפרמטרים, ++C מנצלת את ערכי ברירת המחדל, כדי להשלים את הפרמטר החסר.
- ❖ כאשר בקריאה לפונקציה מושמט פרמטר אחד, מושמטים גם **כל** הפרמטרים הבאים אחריו.

השימוש במנגנון ערכי ברירת מחדל לפרמטרים של הפונקציה מאפשר מצד אחד להגביר את השימושיות החוזרת של הפונקציות (ניצול הגדרת הפונקציה בתוכנית אחת ויותר), ומצד שני - מקל על השימוש בפונקציה עצמה.

הגדרת ערכי ברירת מחדל

קביעת ערכי ברירת מחדל לפונקציה הינה פעולה פשוטה וקלה לביצוע. משתמשים באופרטור ההשמה של שפת ++C, ומציבים את הערך הרצוי לפרמטר בהגדרת הפונקציה, כמו בדוגמה שלהלן.

```
void some_function(int size=12, float cost=19.95)
 {
   // Function statements
 }
```

ערכי ברירת המחדל

בתוכנית **DEFAULTS.CPP** מופיעה הפונקציה show_parameters, שבהגדרתה ניתנים ערכי ברירת מחדל לשלושת הפרמטרים שלה a, b ו- c. בתוכנית קוראים לפונקציה מספר פעמים. בפעם הראשונה לא נקבעים כל פרמטרים, בפעם השנייה נקבע ערך לפרמטר a, בהמשך נקבע ערך לפרמטרים a ו-b, ובסוף נקבע ערך לשלושת הפרמטרים.

```
#include <iostream.h>

void show_parameters(int a=1, int b=2, int c=3)
 {
  cout << "a " << a << " b " << b << " c " << c << endl;
 }

void main(void)
 {
  show_parameters();
  show_parameters(1001);
  show_parameters(1001, 2002);
  show_parameters(1001, 2002, 3003);
 }
```

לאחר הידור התוכנית והרצתה יוצגו על המסך שורות הפלט הבאות:

```
C:\> DEFAULTS  <Enter>
a 1 b 2 c 3
a 1001 b 2 c 3
a 1001 b 2002 c 3
a 1001 b 2002 c 3003
```

ניתן להבחין בתוצאות ההרצה, שהפונקציה משתמשת בערכים הדרושים לה לפי קביעת ערכי ברירת המחדל.

השמטת ערכי פרמטרים

כאשר ערך של פרמטר מסוים אינו מועבר, גם הפרמטרים שבהמשך לא יועברו לפונקציה. במילים אחרות, אין אפשרות **שלא** להעביר פרמטר שהוא במרכז רשימת הפרמטרים של הפונקציה. בתוכנית הקודמת אי העברת ערך לפרמטר b בפונקציה show_parameters, גרם גם לאי העברת ערך לפרמטר c באותה פונקציה. אין כל דרך להעביר ערך לפרמטר c ללא העברת ערך לפרמטר b.

הגדרת ערכי ברירת מחדל לפרמטרים

בעת הגדרת פונקציה, מאפשרת ++C להגדיר ערכי ברירת מחדל לפרמטר אחד או יותר, כדי שאפשר יהיה לנצל אותם, אם בעת הקריאה לפונקציה יושמטו פרמטרים כלשהם. כדי להגדיר ערך ברירת מחדל לפרמטר, יש להשתמש באופרטור ההשמה בהגדרת הפונקציה. לדוגמה, הפונקציה הבאה, payroll, מגדירה ערכי ברירת מחדל לפרמטרים hours ו- rate :

```
float pay(int employ_id, float hours=40, float rate=(5.50)
  {
     // statements
  }
```

כאשר בקריאה לפונקציה מושמט פרמטר אחד, מושמטים בהכרח גם כל הפרמטרים העוקבים לו.

סיכום

בפרק הנוכחי הצגנו כיצד לקבוע ערכי ברירת מחדל לפרמטרים של הפונקציה. בפרקים הבאים נדון בתכונות של ++C כשפה לתכנות מונחה עצמים, וננצל את קביעת ערכי ברירת מחדל של הפרמטרים לאתחול משתנים שונים בתוך המחלקה. לפני שנעבור לפרק הבא מומלץ לבחון אם מובנים הנושאים הבאים :

- ✓ קביעת ערך ברירת מחדל מתבצע על ידי שימוש באופרטור ההשמה להצבת הערכים הרצויים לפרמטרים בכותרת הגדרת הפונקציה.
- ✓ הפונקציה תפעיל את ערך ברירת המחדל של פרמטר כאשר בקריאה לפונקציה לא יועבר ערך לפרמטר הזה.
- ✓ כאשר תוכנית אינה מעבירה ערך של פרמטר, התוכנית חייבת להשמיט את הערכים של כל הפרמטרים העוקבים - תוכניות לא מסוגלות להשמיט ערך של פרמטר מסוים בלבד.

בפרקים הקודמים למדנו שניתן לאחסן במשתנה ערכים המוגדרים לפי טיפוסים int, float וכו'. בפרק 16 נציג כיצד ניתן לשמור מספר ערכים מאותו טיפוס בתוך מערך. למשל, 100 ציונים של תלמידים בכיתה, או המחיר של 50 פריטים. השימוש במערכים והגישה לערכים המאוחסנים בהם, הן משימות קלות מאוד ליישום.

תרגילים

1. כיתבו מחדש את תרגיל 2 מפרק 14, אלא שהפעם, אם הקריאה לפונקציות dec ו-inc אינה מציינת מפורשות את delta, מניחים שערכה הוא 1.
2. כיתבו פונקציה אשר מדפיסה פירמידה, שגובהה וצורת הלבנה שלה (התו ממנו היא מורכבת) הם קלטים לפונקציה. ברירת המחדל היא 10 ו-"*" בהתאמה.
3. כיתבו פונקציה expt אשר מעלה מספר x מטיפוס float בחזקת מספר n מטיפוס unsigned int. אם הקריאה לפונקציה אינה מציינת מפורשות את n, מניחים שערכו של המשתנה הוא 1.

חלק 3

אחסון מידע במערכים ובמבני רשומה

בפרקים הקודמים למדנו שבמשתנה אחד ניתן לאחסן ערך אחד על-פי טיפוס המשתנה. בתוכניות שהצגנו בשני החלקים הראשונים של הספר, הצבנו ערך אחד לכל משתנה שהוגדר. תוכניות מורכבות צריכות בדרך כלל לנהל מידע רב בזמן נתון. למשל, 100 ציונים של תלמידים בכיתה, 20 תוצאות של משחקי כדורגל או שמות וכתובות של 5000 העובדים של מפעל גדול. בחלק הנוכחי של הספר נלמד להשתמש בטיפוסי נתונים שונים כדי לאחסן ערכים שונים באותו משתנה. בהמשך נראה שניצול משתנה אחד לאחסון מספר רב של ערכים הוא מאוד נפוץ ושימושי בתוכניות בשפת ++C.

חלק זה כולל את הפרקים הבאים:

פרק 16

מערכים

בתוכניות שהצגנו בפרקים קודמים אוחסנו נתונים במשתנים במהלך הרצת התוכנית. עד כאן הקצבנו לכל משתנה ערך אחד. בדרך כלל, ברוב התוכניות נדרש טיפול במספר ערכים העונים על אותה הגדרה. למשל, 50 ציונים של תלמידים בכיתה, 100 כותרים של ספרים, או 1000 שמות של קבצים. במקרים אלה ניתן להשתמש במערכים. הגדרת ה**מערך** (**Array**) כוללת את שם המערך, טיפוס המערך ואת מספר הערכים או איברים שניתן לאחסן בו.

בהמשך הפרק נדון בנושאים הבאים :

- **מערך** (Array) הוא מבנה נתונים, המאפשר לאחסן מספר ערכים במשתנה יחיד.
- בעת הצהרה על מערך, יש לציין את טיפוס איברי המערך ואת **מספר הפריטים** (איברים, Array elements) שיאוחסנו בו.
- כל האיברים במערך חייבים להיות מאותו הטיפוס (int ,char וכדי).
- כדי לאחסן ערך במערך, יש לציין את מספר איבר המערך שבו יאוחסן הערך (המספור מתחיל מאפס!).
- כדי לגשת לערך המאוחסן במערך, יש לציין את שם המערך ואת מספר האיבר שבו מאוחסן הערך.
- כאשר מצהירים על מערך, ניתן להשתמש באופרטור ההשמה כדי לאתחל את איברי המערך בערך רצוי כלשהו.
- ניתן להעביר משתנים מטיפוס מערך לפונקציה, כשם שמעבירים לפונקציה כל פרמטר מטיפוס אחר.

בשפת ++C השימוש במערכים הוא נפוץ ומקיף מאוד. בפרק 17 נציג את השימוש במערכים להרכבת מחרוזות של אותיות (כגון כותרים של ספרים, שמות של קבצים וכוי) אשר בהגדרתם הם מערכים של תווים.

הגדרה של משתנה מטיפוס מערך

משתנה מטיפוס **מערך** (**Array**) הוא משתנה המסוגל לאחסן בתוכו מספר ערכים. בדומה לתכונות של המשתנים שהצגנו עד כה, המערכים חייבים להיות מוגדרים לטיפוס נתון מסוים (כגון int, char או float) ולכל מערך חייב להיות שם. בנוסף לתכונות אלו, בהגדרת המערך ניתן להוסיף את מספר הערכים שהוא יהיה מסוגל לאחסן. הערכים שמערך מאחסן חייבים להיות מאותו טיפוס, וזה אומר ש**אין המערך מסוגל לקלוט ערכים מטיפוסים שונים**. בהגדרה הבאה, המערך test_scores מתוכנן לאחסן 100 ציוני מבחנים שונים.

טיפוס המערך

```
int test_scores[100];
```

גודל המערך

לפי הגדרה זו, המהדר של שפת ++C מקצה בזיכרון מקום אשר יספיק לאחסון 100 ערכים מטיפוס int. הערכים שמאוחסנים במערך נקראים **איברי המערך** (**Array elements**).

מערכים מאחסנים ערכים רבים מאותו סוג

כאשר כותבים תוכניות מורכבות, דרוש לפעמים להשתמש במשתנים רבים מאותו טיפוס (type, סוג), כמו למשל מחירים של 50 פריטים, או שמות של 100 עובדים. במקום לעבוד עם מספר גדול של משתנים, אשר לכל אחד מהם שם נפרד ושונה, ניתן להגדיר מערך, שהוא משתנה אחד, והוא זה שמכיל מספר ערכים הקשורים זה לזה.

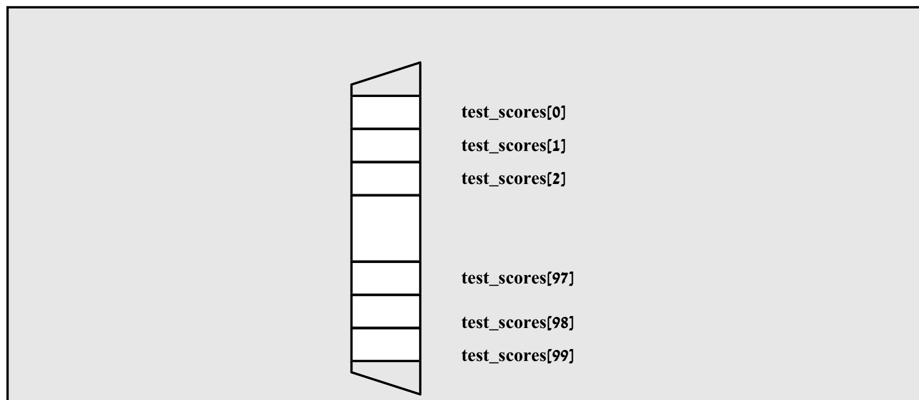
כדי להגדיר מערך, צריך לציין את סוגו (Type) ואת מספר האיברים שיכיל, ולתת לו שם ייחודי. לדוגמה, המשפטים הבאים מגדירים שלושה מערכים שונים:

```
float part_cost [50];
int employee_age [100];
float stock_prices [25];
```

גישה לאיברי המערך

במשתנה מסוג מערך ניתן לאחסן מספר ערכים. כדי לגשת לערך מסוים במערך נעזרים ב**ערך אינדקס** (**Index value**), המצביע על האיבר הרצוי במערך.

למשל, כדי לגשת לאיבר הראשון של המערך test_score ערך האינדקס שעלינו להשתמש בו הוא 0. האינדקס של האיבר השני הוא הערך 1, האינדקס של האיבר השלישי הוא הערך 2 וכך הלאה. בתרשים 16.1 מתוארת שיטת האינדקס של המערך כאשר האינדקס של האיבר הראשון במערך הוא 0 והאינדקס של האיבר האחרון במערך שווה לגודל המערך (מספר האיברים) פחות 1.



תרשים 16.1: סימון האינדקס של איברי המערך.

יש להדגיש, כי בשפת ++C תמיד ערך האינדקס של האיבר הראשון במערך הוא 0, והאינדקס של האיבר האחרון הוא גודל המערך פחות 1.

בתוכנית **ARRAY.CPP** מוגדר מערך בשם values המתוכנן לאחסון חמישה ערכים מטיפוס int. בתוכנית מתבצעת השמה של הערכים 100, 200, 300, 400 ו- 500 לחמשת האיברים של המערך.

```
#include <iostream.h>

void main(void)
 {
   int values[5];   // Declare the array

  values[0] = 100;
  values[1] = 200;
  values[2] = 300;
  values[3] = 400;
  values[4] = 500;

  cout << "The array contains the following values" << endl;
  cout << values[0] << ' ' << values[1] << ' ' << values[2]
            << ' '<< values[3] << ' ' << values[4] << endl;
 }
```

האיבר הראשון במערך (הערך 100) מוגדר בכניסה [0] **אפס** (values[0]). האיבר האחרון (הערך 500) מוגדר בכניסה [4] **ארבע** (גודל המערך (5) פחות 1).

שימוש בערך אינדקס כדי לגשת לאיברי המערך

המערך, כפי שכבר למדנו, מאפשר לאחסן מספר רב של ערכים באותו משתנה (משתנה המערך). כדי לפנות אל ערכים כלשהם במערך, התוכנית צריכה להשתמש בערכי אינדקס. ערך אינדקס (Index value) מציין את המספר הסידורי של איבר המערך שאליו המתכנת רוצה לפנות לקריאה או כתיבה. לדוגמה, המשפט הבא מקצה את הערך 100 לאיבר הראשון במערך, אשר שמו הוא scores:

```
scores [0] = 100;
```

שים לב! *המספור תמיד מתחיל מ- 0!*

בעת ההצהרה על מערך מגדירים את מספר האיברים שיכיל. המשפט הבא מצהיר על מערך מטיפוס int בעל 100 איברים:

```
int scores [100];
```

במקרה זה, איברי המערך הם scores [0] עד scores [99].

משתנה אינדקס

הדרך הנפוצה לביצוע פעולות במערכים של התוכניות היא להיעזר במשתנה המשמש כאינדקס לאיברי המערך. למשל, המשפט הבא יגרום להצבת הערך 400 באיבר values[3], בהנחה שבמשתנה i מאוחסן הערך 3.

```
values[i] = 400;
```

בתוכנית הבאה, **SHOWARRA.CPP**, נעזרים במשתנה i כמונה לולאת for וכאינדקס המאפשר הצגת הערך של איברי המערך במסך. הערך ההתחלתי שהלולאה מציבה במשתנה i הוא 0, כך שהוא מתייחס לאיבר הראשון של המערך values[0]. הלולאה מסתיימת כאשר הערך של i גדול מ- 4 (המתייחס לאיבר האחרון במערך).

```
#include <iostream.h>

void main(void)
 {
   int values[5];   // Declare the array
   int i;

   values[0] = 100;
   values[1] = 200;
   values[2] = 300;
   values[3] = 400;
   values[4] = 500;

   cout << "The array contains the following values" << endl;

   for (i = 0; i < 5; i++)
     cout << values[i] << ' ';
 }
```

בכל פעם שהלולאה מקדמת את ערך המשתנה i, מתאפשרת הגישה לאיבר הבא במערך. מומלץ להריץ שוב את התוכנית ולערוך את השינויים הבאים ללולאת for:

```
for (i = 4; i >=  0; i-)
  cout << values[i] << ' ';
```

במקרה זה, התוכנית תציג את הערכים המאוחסנים במערך, מהאיבר האחרון ועד לאיבר הראשון.

קביעת ערכי המערך במשפט ההגדרה

כבר ראינו שבשפת ++C ניתן לקבוע ערך התחלתי למשתנים במשפטי ההגדרה שלהם. הדבר נכון גם בקשר למערכים. אתחול המערכים במשפט ההגדרה שלהם מתבצע על ידי רישום ערכים של המערך בתוך סוגריים מסולסלים, כאשר בין הגדרת המערך והסוגריים כותבים את הסימן שווה. לדוגמה, המשפט הבא מבצע אתחול של המערך values: קביעת ערכים לאיברי המערך.

```
int values[5] = { 100, 200, 300, 400, 500 };
```

בצורה דומה, המשפט הבא מבצע אתחול של מערך מטיפוס float:

```
float salaries[3] = { 25000.00, 35000.00, 50000.00 };
```

רוב המהדרים של שפת ++C מציבים את הערך אפס לאיברים שלא מציינים עבורם ערך התחלתי כלשהו במשפט ההגדרה. לדוגמה, במשפט הבא מתבצע אתחול של שלושת האיברים הראשונים של המערך המונה חמישה איברים:

```
int values[5] = { 100, 200, 300 };
```

האיברים values[3] ו- values[4] אינם מקבלים ערך התחלתי כלשהו.

על ידי אתחול המערך במשפט ההגדרה ניתן לקבוע את גודלו, ללא ציון מפורש של הגודל. המהדר מכין מקום בזיכרון לאחסון מספר האיברים המצוינים בתוך הסוגריים המסולסלים. ההגדרה הבאה למשל, יוצרת מערך שגודלו מתאים לאחסון ארבעה ערכים מטיפוס int:

```
int numbers[] = { 1, 2, 3, 4 };
```

העברת מערכים לפונקציות

כשם שאפשר להעביר לפונקציה ערכים של משתנים, אפשר להעביר לה גם מערכים. הפונקציה יכולה לאתחל מערך, לעדכן את הערכים המאוחסנים בו, או להציג אותם על המסך. הפונקציה מקבלת את כתובת המערך ולכן היא מתייחסת למערך המקורי. להעברת המערך לפונקציה צריך לציין את טיפוס המערך, אך לא את גודלו. בדרך כלל המערך מועבר לפונקציה כפרמטר, לפי התחביר המוצג בדוגמה שלהלן.

```
void some_function(int array[], int number_of_elements);
```

בתוכנית **ARRAYFUN.CPP** הפונקציה show_array מקבלת כפרמטר את המערך ומציגה את ערכי האיברים שלו על המסך. כאן המקום להזכיר, כי בשפת C++ ההפרדה בין משפטים נקבעת על ידי נקודה-פסיק. לכן, פיצול שורה או הוספת רווחים בין הפקודות שבמשפט לא קובעים את סיומה ולא משפיעים על מהלך הרצת התוכנית. בדרך זו ניתן לפצל את המשפטים למספר שורות כדי שהתוכנית תהיה ברורה וקריאה יותר.

```
#include <iostream.h>

void show_array(int array[], int number_of_elements)
 {
   int i;

   for (i = 0; i < number_of_elements; i++)
     cout << array[i] << ' ';

   cout << endl;
 }

void main(void)
 {
   int little_numbers[5] = { 1, 2, 3, 4, 5 };
   int big_numbers[3] = { 1000, 2000, 3000 };

   show_array(little_numbers, 5);
   show_array(big_numbers, 3);
 }
```

במשפט הקריאה לפונקציה, שבדוגמה הבאה, המערך מועבר על ידי ציון שמו:

```
show_array(little_numbers, 5);
```

בתוכנית **GETARRAY.CPP**, תפקידה של הפונקציה get_values הוא להציב ערך לשלושת האיברים הראשונים של המערך numbers.

```
#include <iostream.h>

void get_values(int array[], int number_of_elements)
 {
   int i;

   for (i = 0; i < number_of_elements; i++)
    {
      cout << "Enter value " << i << ": ";
      cin >> array[i];
    }
 }

void main(void)
 {
   int numbers[3];

   get_values(numbers, 3);

   cout << "The array values are as follows" << endl;

   for (int i = 0; i < 3; i++)
     cout << numbers[i] << endl;
 }
```

המערך מועבר לפונקציה על ידי ציון שמו. הפונקציה מציבה ערכים במערך: הלולאה מציבה ערך לאיבר התורן. בפרק 20 נלמד ששפת ++C מעבירה לפונקציה את המערכים על ידי שימוש במצביעים. לכן ניתן לשנות את ערך האיברים של מערך בתוך הפונקציה.

סיכום

בפרק זה למדנו שניתן לאחסן מספר ערכים, מאותו סוג במשתנה אשר מוגדר כמערך. בשפת ++C יש שימוש רב במערכים. לפני שנעבור לפרק הבא מומלץ לבחון אם מובנים הנושאים הבאים:

- ✓ המערך הוא משתנה המסוגל לאחסן מספר ערכים מאותו טיפוס.
- ✓ משפט הגדרת המערך כולל את טיפוס המערך, שם המערך ומספר הערכים שיאוחסנו בו.
- ✓ הערכים שמאוחסנים במערך הם איברי המערך (Array elements).
- ✓ הערך הראשון של המערך מאוחסן באיבר אפס (array[0]). הערך אחרון מאוחסן באיבר האחרון של המערך, שמספרו נקבע לפי גודל המערך פחות 1.

- ✓ בדרך כלל נעזרים במשתני אינדקס לשם גישה לאיברי המערך.
- ✓ כאשר מעבירים מערך כפרמטר לפונקציה, יש לציין בפונקציה את טיפוס המערך ואת שמו. אין צורך לציין את גודל המערך.

בפרק 17 נציג את המחרוזות וכיצד לעבוד איתן. נראה שמחרוזת היא על פי ההגדרה מערך מטיפוס char, כלומר - **מחרוזת תווים (Character string)**.

תרגילים

1. כיתבו פונקציה המקבלת מערך של שלמים שאורכו 10, ומחזירה את ערכו של המספר הקטן ביותר במערך. כיתבו בנוסף שיגרה main, המפעילה את השיגרה הזו ומדפיסה את התוצאה.
2. כיתבו תוכנית המקבלת מהמשתמש מספרים, עד אשר יקיש את המספר 1-. כאשר הוקש מספר זה, יש להדפיס את כל הספרות שנקלטו בסדר הפוך (הניחו שהמשתמש מקיש פחות מ- 10 מספרים).
3. כיתבו תוכנית המקבלת מהמשתמש מספרים בטווח 0 עד 49, עד אשר הוא מקליד ערך שנמצא מחוץ לטווח. כעת יש להדפיס עבור כל ערך בתחום את מספר הפעמים שהמשתמש הקליד אותו.
4. כיתבו פונקציה המקבלת שני מערכים של עשרה ערכים שלמים (int). מניחים שכל אחד מהמערכים הינו ממוין. השיגרה צריכה להדפיס בצורה ממוינת את כל המספרים המופיעים בשני המערכים.

פרק 17

מחרוזות תווים

מחרוזות (**Strings**) הן מבנים המתאימים לאחסון מידע כגון שמות קבצים, כותרות של ספרים, שמות העובדים וצירופי אותיות אחרים. ברוב התוכניות הכתובות בשפת C++ יש שימוש במחרוזות. בהמשך הפרק נראה ש**מחרוזות התווים** (**Characters strings**) הן מערכים מטיפוס char המסתיימים בתו המיוחד NULL (ASCII 0). בפרק זה נבחן בפירוט מחרוזות תווים. נלמד לאחסן ולטפל במחרוזות תווים כדי להשיג יתרונות הנובעים משימוש בפונקציות של ספריות ההרצה (Runtime libraries).

בהמשך הפרק נדון בנושאים הבאים:

- הגדרת מחרוזת בתוכנית - הגדרה של מערך מטיפוס char.
- השמה של תווים במחרוזת - השמה של תווים באיברי המערך.
- חשיבות התו המיוחד NULL (ASCII 0), כדי לציין את התו האחרון במחרוזת.
- אתחול של מחרוזת יכול להעשות בעת ההגדרה שלה.
- העברת מחרוזות לפונקציות, בדרך כלל דומה להעברת כל מערך אחר.
- שימוש בפונקציות של ספריות ההרצה run-time כדי לטפל במחרוזות תווים.

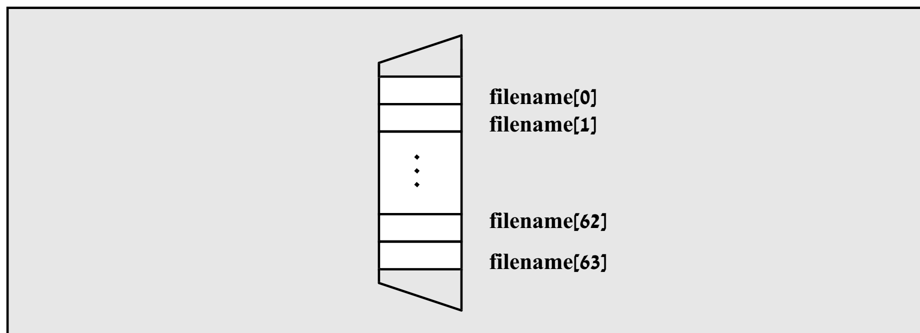
בשפת C++ המחרוזות מאוחסנות במערכים של תווים. ברוב התוכניות הכתובות בשפת C++ יש שימוש במחרוזות, ועל כן מומלץ להריץ את כל התוכניות שבפרק זה, כדי לתרגל את השימוש במחרוזות. ראוי לשים לב לכך שעבודה במחרוזות תווים דומה לעבודה במערכים, כפי שראינו בפרק 16.

הגדרת מחרוזת בתוכנית

בתוכניות הכתובות בשפת C++ מנצלים את תכנות המחרוזות לאחסון שמות אנשים, שמות של קבצים ומידע אחר המבוסס על תווים ואותיות. הגדרת המחרוזת נעשית על ידי הגדרת מערך מטיפוס char שגודלו מתאים לדרישות התוכנית. למשל, התוצאה של ההגדרה הבאה היא יצירת משתנה מטיפוס מחרוזת בשם filename שמסוגל לאחסן 64 תווים (יש לזכור שהתו המיוחד NULL הוא חלק מתוך 64 התווים).

```
char filename[64];
```

בתרשים הבא (17.1) מוצגת בצורה גרפית ההגדרה, כאשר אינדקס המערך מתחיל באיבר filename[0] ומסתיים באיבר filename[63].



תרשים 17.1: מחרוזת בשפת C++ היא מערך מטיפוס char.

הדבר הראשון המבדיל בין המערכים הרגילים ובין המחרוזות בשפת C++ הוא אופן סימון סוף המחרוזת. בשפת C++ סוף המחרוזת מצוין על ידי התו המיוחד NULL שהוא תו ASCII שערכו 0. הוא מוגדר בשפת C++ כתו מיוחד '\0'. לכן, לאחר השמה של תווים במחרוזת יש להציב את התו NULL ('\0') לאחר התו האחרון של המחרוזת.

לדוגמה, בתוכנית **ALPHABET.CPP** נעזרים בלולאת for כדי להציב במשתנה alphabet את אותיות האלף-בית הלועזי (מ- A עד Z). בהמשך, התוכנית מציבה בסיום המחרוזת את התו NULL, ומציגה על המסך את תוכן המחרוזת תוך שימוש בפקודה cout.

```
#include <iostream.h>

void main(void)
 {
   char alphabet[27];  // 26 letters plus NULL
   char letter;
   int index;
```

```
   for (letter = 'A', index = 0; letter <= 'Z';
     letter++, index++)
     alphabet[index] = letter;

   alphabet[index] = NULL;
   cout << "The letters are " << alphabet;
 }
```

יש לשים לב שהתוכנית מציבה את התו NULL בסיום המחרוזת, וכך היא מציינת בה את האיבר האחרון שלה.

```
alphabet[index] = NULL;
```

כדי להציג מחרוזת על המסך, משתמשים בפקודה cout, אשר מנתבת את המחרוזת אל ערוץ הפלט, עד לתו NULL.

כעת נתבונן בלולאה for של התוכנית. ניתן לראות שהלולאה קובעת ערך התחלתי לשני משתנים: letter ו-index ומקדמת את ערכם. כאשר משתמשים בלולאה for בשני משתני עזר, יש להפריד ביניהם בפסיק.

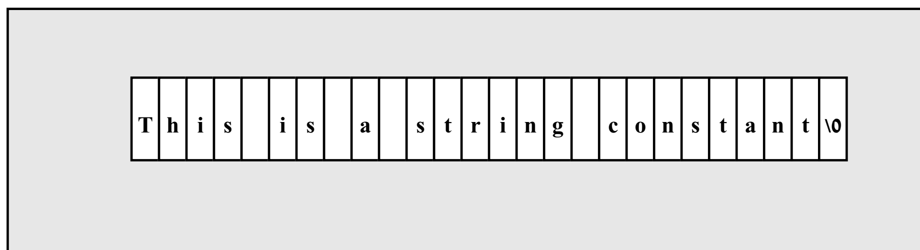
```
for (letter = 'A', index = 0; letter <= 'Z';
  letter++, index++)
```

הצבה אוטומטית של NULL למחרוזות בשפת ++C

בכל התוכניות בספר זה הצבנו מחרוזות תווים בין גרשיים, כמו בדוגמה זו:

```
"This is a string constant"
```

כאשר מגדירים מחרוזת, שפת ++C מציבה בצורה אוטומטית את תו NULL בסיומה, כפי שמוצג בתרשים הבא.



תרשים 17.2: הצבה אוטומטית של התו NULL בסיום המחרוזת.

התו NULL, שהמהדר מציב בסוף המחרוזת קובע לפקודה cout את סיום הצגת המחרוזת שהועברה לערוץ הפלט.

התו המיוחד NULL

מחרוזת מוגדרת כמערך של תווים שבסיומו נמצא התו NULL ('\0'). הגדרת המחרוזת בתוכנית היא למעשה הגדרת מערך מטיפוס char. במהלך התוכנית מתבצעת פעולת השמה של תווים למחרוזת, והתוכנית מוסיפה למחרוזת את התו NULL המסמן את סוף המחרוזת. לעומת זאת, כאשר מגדירים מחרוזת על ידי הכנסת התווים בתוך גרשיים כפולים, מהדר ++C מציב אוטומטית את התו NULL בסוף המחרוזת. רוב הפונקציות בשפת ++C מתבססות על העובדה שהתו NULL מסמן את סוף המחרוזת.

התוכנית הבאה, **LOOPNULL.CPP**, משנה במקצת את התוכנית הקודמת, על ידי שימוש בלולאת for להצגת תכולת המחרוזת:

```
#include <iostream.h>

void main(void)
 {
  char alphabet [27];   // 26 characters plus NULL
  char letter;
  int index;

  for (letter = 'A', index = 0; letter <= 'Z'; letter++,
                                               index++)
    alphabet [index] = letter;

  alphabet [index] = NULL;

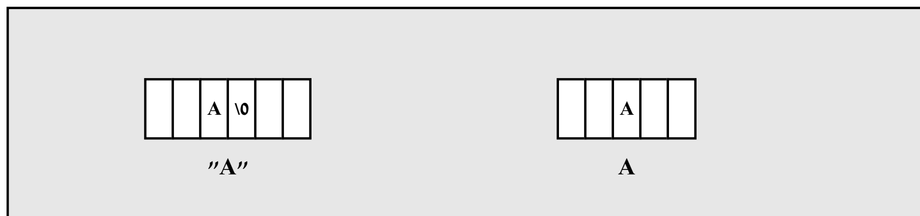
  for (index = 0; alphabet [index] != NULL; index++)
    cout << alphabet [index];

  cout << endl;
 }
```

כפי שניתן לראות, לולאת for בוחנת את התווים במחרוזת, זה אחר זה. אם התו אינו NULL (כלומר, מסמן את התו האחרון במחרוזת), הלולאה מציגה את התו, מקדמת את האינדקס המציין את המספר הסידורי במערך, וממשיכה את התהליך.

ההבדל בין 'A' לבין "A"

חשוב להבין מה הוא השוני מבחינת ++C בין תו הנמצא בתוך גרשים בודדים (דוגמת 'A') לבין אותו תו, כאשר הוא נמצא בתוך גרשים כפולים (דוגמת "A"). התו הנמצא בתוך גרשים בודדים הינו **תו קבוע** (**Character constant**) מטיפוס char. המהדר מקצה עבורו שטח זיכרון שגודלו בית אחד, המתאים לאחסון קבוע מסוג תו. במקרה השני, כאשר התו נמצא בין גרשים כפולים, לפנינו **קבוע מחרוזת** (**String constant**) שמכיל את התו עצמו ואת התו NULL שנוסף אוטומטית על ידי המהדר כדי לציין את סוף המחרוזת. לכן, המהדר מקצה שטח שגודלו שני בתים בזיכרון המחשב לאחסון מחרוזת תו. בתרשים 17.3 מתואר כיצד מהדר שפת ++C מאחסן בזיכרון קבוע מטיפוס תו 'A' וקבוע מטיפוס מחרוזת "A".



תרשים 17.3: אופן אחסון קבוע מטיפוס תו וקבוע מטיפוס מחרוזת.

אתחול המחרוזת

בפרק 16 הצגנו את אופן אתחול המערכים במשפט ההגדרה שלהם. מבחינה זו, אתחול המחרוזות אינו שונה. כדי לקבוע ערך התחלתי למחרוזת תווים במשפט ההגדרה, יש לכתוב את המחרוזת הרצויה בתוך סוגריים כפולים, כדלקמן:

```
char title[64] = "Rescued by C++";
```

כאשר מספר התווים הרשומים במחרוזת קטן מגודל המחרוזת המוגדר, המהדר מציב את התו NULL לשאר התווים במחרוזת. לעומת זאת, כאשר בהגדרת המחרוזת לא מציינים את גודל המחרוזת, המהדר יקצה לה שטח זיכרון, המתאים לאחסון מספר התווים הדרוש ועוד התו NULL.

```
char title[] = "Rescued by C++";
```

בתוכנית **INIT_STR.CPP** מתבצע אתחול של מחרוזות במשפטי הגדרתן.

```
#include <iostream.h>

void main(void)
 {
   char title[64] = "Rescued by C++";
   char lesson[64] = "Understanding Character strings";
   cout << "Book: " << title << endl;
   cout << "Lesson: " << lesson << endl;
 }
```

ברוב התוכניות המוצגות בהמשך הספר נאתחל את המחרוזות במשפט ההגדרה שלהן.

העברת מחרוזות לפונקציות

העברת מחרוזת כפרמטר לפונקציה נעשית בצורה דומה להעברת מערך רגיל. בהגדרת הפונקציה יש להגדיר את הפרמטר מטיפוס char ולהציב אחרי שמו סוגריים מרובעים. אין צורך לציין את גודל המחרוזת המתקבלת. לדוגמה, בתוכנית **SHOW_STR.CPP** מוגדרת פונקציה שתפקידה להציג מחרוזת כלשהי על המסך.

```
#include <iostream.h>

void show_string(char string[])
 {
   cout << string << endl;
 }

void main(void)
 {
   show_string("Hello, C++!");
   show_string("I've been Rescued by C++");
 }
```

ניתן לראות שהפונקציה show_string מתייחסת לפרמטר המחרוזת כאל מערך רגיל.

```
void show_string(char string[])
```

בפרק זה למדנו שהפונקציות בשפת C++ נעזרות בתו NULL כדי לזהות את סוף המחרוזת. בתוכנית **STR_LEN.CPP** מוגדרת פונקציה string_length אשר סורקת את המחרוזת שהיא מקבלת כפרמטר עד לתו NULL, וכך היא מחשבת את גודל המחרוזת. במהלך התוכנית מועברות לפונקציה מספר מחרוזות שונות.

```
#include <iostream.h>

int string_length(char string[])
 {
  int i;

  for (i = 0; string[i] != '\0'; I++); //Do nothing
                                       // but loop to next
                                       // character

  return(i);  // The length of the string
 }

void main(void)
 {
  char title[] = "Rescued by C++";
  char lesson[] = "Understanding Character Strings";

  cout << "The string " << title << "contains " <<
          string_length(title) << " characters" << endl;

  cout << "The string " << lesson <<"contains " <<
          string_length(lesson) << "characters" << endl;
 }
```

פונקציות רבות בשפת C++ סורקות מחרוזות עד למציאת התו NULL בדרך שהצגנו בתוכנית הקודמת.

ניצול העובדה שערך קוד ASCII של התו NULL הוא 0

בפרק 7 למדנו שערך 0 מייצג מצב של תנאי שקרי (שאינו מתקיים, false) ולפיכך, מכיון שהתו NULL מסומל בקוד ASCII על ידי הערך 0, ניתן לפשט את פעולתן של לולאות רבות: פונקציות רבות סורקות מחרוזות בשיטת תו אחר תו, כדי למצוא את התו NULL. לדוגמה, לולאת for הבאה ממחישה כיצד פונקציה עשויה לחפש תו NULL במחרוזת:

```
for (index = 0; string [index] != NULL; index++)
  ;
```

מכיון שערך התו NULL הוא 0, לולאות רבות המחפשות את התו הזה במחרוזת ניתנות לפישוט באופן הבא:

```
for (index = 0; string [index]; index++)
    ;
```

במקרה זה, כל עוד הביטוי string[index] אינו NULL (הערך 0 מציין תשובה שלילית, מצב שקרי), הלולאה ממשיכה לפעול.

פונקציות מחרוזת של ספריית ההרצה הסטנדרטית

בפרק 11 למדנו שרוב המהדרים של שפת ++C כוללים אוסף של פונקציות רבות עוצמה הנקרא **ספריית הרצה סטנדרטית** (**Run-time library**), או בפשטות: **פונקציות ספריה**. בספריה זו נמצאות פונקציות רבות, המטפלות במחרוזות. לדוגמה, הפונקציה strupr ממירה את האותיות הלועזיות הקטנות לאותיות גדולות. פונקציה נוספת היא strlen, המחזירה את מספר התווים שבמחרוזת מסוימת. בנוסף, קיימות פונקציות המאפשרות לסרוק מחרוזת ולמצוא בתוכו תו מסוים. לדוגמה, בתוכנית הבאה, **STRUPR.CPP**, מתואר השימוש בפונקציות: strupr ו-strlwr שבספריית ההרצה.

```
#include <iostream.h>
#include <string.h> // Contains function prototypes

void main(void)
 {
   char title[] = "Rescued by C++";
   char lesson[] = "Understanding Character Strings";

   cout << "Uppercase: " << strupr(title) << endl;
   cout << "Lowercase: " << strlwr(lesson) << endl;
 }
```

השימוש בפונקציות ספריית ההרצה חוסך זמן רב בפיתוח התוכניות.

יש לציית לכללים ונהלים

כפי שלמדנו, פונקציות המטפלות במחרוזות תווים, מצפות למצוא תו NULL, אשר מסמן את התו האחרון במחרוזת. לכן, עליך לוודא כי תוכניותיך אכן דואגות להוסיף בסוף מחרוזות התווים שהן יוצרות את התו הזה. אם לא כך, פונקציות המסתמכות על המוסכמה הזו תיכשלנה בפעולתן.

סיכום

ברוב התוכניות בשפת ++C יש שימוש במחרוזות תווים. בפרק זה למדנו כיצד לטפל במחרוזות. בפרק הבא נלמד לאחסן נתונים מטיפוסים שונים במבנה מוגדר - structure. כך למשל במבנה, או מבנה רשומה, מכילים את כל נתוני העובד: שם, כתובת, שנת לידה, מצב משפחתי, שכר ועוד.

לפני שנעבור לפרק הבא, מומלץ לבחון אם מובנים הנושאים הבאים:

- ✓ מחרוזת תווים מוגדרת כמערך של תווים אשר מסתיים בתו ASCII 0, שהוא התו NULL.
- ✓ יצירת מחרוזת תווים נעשית על ידי הגדרת מערך מטיפוס char.
- ✓ הצבת התו NULL בסוף המחרוזת מוטלת על התוכנית (לאחר התוכן האחרון).
- ✓ מהדר שפת ++C מציב את התו NULL בצורה אוטומטית לקבועים מטיפוס מחרוזת, המוגדרים בין גרשיים.
- ✓ אתחול מחרוזת במשפט ההגדרה נעשה על ידי כתיבת תוכן המחרוזת בין גרשיים.
- ✓ רוב המהדרים בשפת ++C כוללים מספר רב של פונקציות ספריה (Run-time library), המטפלות במחרוזות.

תרגילים

1. כיתבו תוכנית המגדירה שתי מחרוזות (מערכי תווים). האחת מאותחלת ל-"I am a string" והשנייה - ל-"I am a string too". ההגדרה של המחרוזת הראשונה תציין במפורש את אורך המחרוזת. ההגדרה השנייה לא תעשה זאת. התוכנית תדפיס את המחרוזות למסך.

2. כיתבו פונקציה אשר מקבלת מחרוזת ותו. אם התו מופיע במחרוזת, הפונקציה מחזירה 1, ואחרת - 0.

3. כיתבו פונקציה אשר מקבלת מחרוזת ותו, ומחזירה את מספר הפעמים שהתו מופיע במחרוזת.

4. כיתבו פונקציה אשר מקבלת שתי מחרוזות ומחזירה את מספר המקומות (האינדקסים) בהם המחרוזות זהות. למשל, המחרוזות a**b**c**d** ו- b**bad** זהות בשני מקומות.

פרק 18

מבני רשומות

בפרק 16 למדנו שבשפת ++C ניתן לאחסן מידע מאותו טיפוס בתוך מערך. הראינו שהשימוש במערכים נוח מאוד ומועיל. אולם, במקרים רבים יש צורך לאחסן מידע המורכב מנתונים שאינם מאותו טיפוס, כמו למשל, בתוכנית המטפלת ברשומות עובדים. בתוכנית כזו יש צורך לייחס לעובד מסוים את שם העובד, גילו, משכורתו, כתובתו, מספר עובד ומידע נוסף. כדי לאחסן מידע מגוון זה, צריך להשתמש בטיפוסים שונים של משתנים: int, real, float ומחרוזות. כדי לאחסן מידע מסוג זה, משתמשים **במבנה** (**Structure**), או **מבנה רשומה** (**Record structure**). נלמד שמבנה הרשומה הוא משתנה המכיל בתוכו יחידות מידע הקרויות **שדות** (members, או fields), והן מטיפוסים שונים.

בהמשך הפרק נדון בנושאים הבאים:

- **מבני רשומה** (Structures) מאפשרים לקבץ בתוך משתנה אחד נתונים מטיפוסים שונים, אשר קשורים זה לזה.
- מבנה רשומה מורכב משדה נתונים אחד או יותר (שדות הרשומה, members).
- מבנה רשומה מוגדר על ידי ציון שם המבנה והשדות המרכיבים אותו.
- בהגדרת הרשומה מציינים את טיפוסי השדות (משתנים), כמו למשל, int, char או float. יש לתת שם ייחודי לכל שדה.
- לאחר שמגדירים מבנה רשומה ניתן להצהיר בתוכנית על משתנים מטיפוס המבנה (Structure type).
- כדי שפונקציה תוכל לשנות שדות של מבנה רשומה, יש להעביר לה את כתובת המבנה על פי כתובת.

מבנה הרשומה הוא אבן היסוד של מבנה המחלקה **בתכנות מונחה-עצמים**, כפי שנלמד בחלק 4 של הספר. על כן מומלץ מאוד להשקיע זמן להבנת מבנה הרשומה ולהריץ שוב ושוב את התוכניות המוצגות בפרק זה.

הגדרת מבנה רשומה

המבנה (**Structure**), מתאר את **תבנית הרשומה** (**Template**) שאחר כך אפשר להצהיר עליה כמשתנה. במילים אחרות, קודם מגדירים כמבנה הרשומה ולאחר מכן מגדירים את המשתנים מטיפוס הרשומה. כדי להגדיר רשומה יש להשתמש במילה השמורה struct ובהמשכה לכתוב את שם המבנה וסוגר מסולסל שמאלי. בין הסוגר המסולסל השמאלי לבין הסוגר המסולסל הימני מגדירים את **שדות הרשומה** (**Members**) ואת הטיפוס שלהם. אחרי הגדרת הרשומה ניתן להגדיר משתנים מטיפוס הרשומה:

```
struct name {
  int member_name_1;
  float member_name_2;
} variable;
```

הגדרת שדות (Members)

הגדרת משתנה

לדוגמה, הגדרת מבנה הרשומה הבא מתאים לאחסון מידע אודות עובד:

```
struct employee {
  char name[64];
  long employee_id;
  float salary;
  char phone[10];
  int office_number;
};
```

במקרה הקודם לא מוגדר משתנה מטיפוס הרשומה. בשפת ++C ניתן להגדיר משתנים מטיפוס רשומה רק לאחר שמוגדר מבנה הרשומה ולהשתמש לצורך זה **בשם המבנה** (שלעיתים קרוי **Structure tag**). להלן דוגמה:

שם המבנה (Tag)

```
employee boss, worker, new_employee;
```

הגדרת משתנים

במשפט הקודם הגדרנו שלושה משתנים מטיפוס מבנה הרשומה employee. במקרים מסוימים, הגדרת משתנים מטיפוס רשומה נעשית תוך שימוש במילה השמורה **struct**, על פי הדוגמה הבאה:

```
struct employee boss, worker, new_employee;
```

להגדרת משתנה רשומה בשפת C חייבים להשתמש במילה השמורה struct: לכן, מטעמי הרגל נוהגים מתכנתי C להציב מילה זו לפני הגדרת המשתנים. בכל מקרה, בשפת ++C השימוש במילה השמורה sturct להגדרת משתנה אינו חובה.

שדות הרשומה

מבנה הרשומה מורכב מיחידות מידע הנקראות **שדות** (fields או members). כדי לאחסן מידע בשדות וכדי לגשת למידע המאוחסן בהם משתמשים **באופרטור הנקודה** (**.**). לדוגמה, במשפטים הבאים מתבצעת השמה של ערכים בשדות שונים של המשתנה worker שהוא מטיפוס מבנה הרשומה employee.

```
worker.employee_id = 12345;
worker.salary = 25000.00;
worker.office_number = 102;
```

הגישה לשדה של משתנה הרשומה מתבצעת על ידי ציון שם המשתנה ושם השדה הרצוי, כאשר הם מופרדים על ידי נקודה. בתוכנית הבאה, **EMPLOYEE.CPP**, מתואר השימוש במבנה הרשומה employee.

```
#include <iostream.h>
#include <string.h>

void main(void)
 {
  struct employee {
     char name[64];
     long employee_id;
     float salary;
     char phone[10];
     int office_number;
  } worker;

  // Copy a name to the string
  strcpy(worker.name, "John Doe");

  worker.employee_id = 12345;
  worker.salary = 25000.00;
  worker.office_number = 102;

  // Copy a phone number to the string
  strcpy(worker.phone, "555-1212");

  cout << "Employee: " << worker.name << endl;
  cout << "Phone: " << worker.phone << endl;
  cout << "Employee id: " << worker.employee_id << endl;
  cout << "Salary: " << worker.salary << endl;
  cout << "Office: " << worker.office_number << endl;
 }
```

ההשמה של ערכים לשדות מטיפוס int ו- float מתבצעת בצורה ישירה. לעומת זאת, כדי להציב ערכים למחרוזות name ו- phone משתמשים בפונקציה strcpy. הצבת ערכים לשדה מחרוזת מתבצעת על ידי העתקה של מחרוזות (למעט אתחול המחרוזת בעת הגדרת המשתנה).

הגדרת משתנה הרשומה

מבנה הרשומה בשפת ++C מאפשר לכלול בתוך שם משתנה אחד מידע המורכב מטיפוסים שונים של נתונים. מבנה מוגדר על ידי תבנית המידע המאפשר להגדיר משתנים מטיפוס הרשומה בשלבים מאוחרים בתוכנית. כל תבנית מקבלת שם ייחודי בתוכנית. על ידי שימוש בשם התבנית ניתן להגדיר משתנים מטיפוס רשומה זו. רכיביה של רשומה נקראים שדות. בשפת ++C ניתן לגשת לשדות הרשומה או לאחסן בהן מידע על ידי שימוש באופרטור הנקודה, כפי שמתואר בדוגמה הבאה:

```
variable.member = some_value;
some_variable = variable.other_member;
```

מבנה הרשומה והפונקציות

בפונקציות שלא מתבצע בהן שינוי בערכי שדות הרשומה ניתן להעביר את שם משתנה הרשומה כפרמטר לפונקציה. בתוכנית **SHOW_EMP.CPP** מוגדרת הפונקציה show_employee אשר תפקידה להציג את ערכי שדות הרשומה מטיפוס employee.

```
#include <iostream.h>
#include <string.h>

struct employee {
  char name[64];
  long employee_id;
  float salary;
  char phone[10];
  int office_number;
 };

void show_employee(employee worker)
 {
   cout << "Employee: " << worker.name << endl;
   cout << "Phone: " << worker.phone << endl;
   cout << "Employee id: "<< worker.employee_id << endl;
   cout << "Salary: " << worker.salary << endl;
   cout << "Office: " << worker.office_number << endl;
 }
```

```
void main(void)
 {
   employee worker;

   // Copy a name to the string
   strcpy(worker.name, "John Doe");

   worker.employee_id = 12345;
   worker.salary = 25000.00;
   worker.office_number = 102;

   // Copy a phone number to the string
   strcpy(worker.phone, "555-1212");
   show_employee(worker);
 }
```

בתוכנית הקודמת הוגדר המשתנה worker מטיפוס רשומה. משתנה זה מועבר לפונקציה show_employee על ידי ציון שם המשתנה בלבד. בתוך הפונקציה מתבצעת הצגת שדות הרשומה על המסך. יש לשים לב, שבתוכנית הרשומה employee מוגדרת מחוץ לתוכנית הראשית main, ומחוץ לפונקציה show_employee. הגדרת הרשומה emloyee חייבת להיות לפני הגדרת הפונקציה show_employee, מכיון שבפונקציה מוגדר משתנה worker שהוא מטיפוס הרשומה employee.

הערה


יש לשים לב שצריך להעביר את המבנה לפונקציה. על כן מומלץ להעביר לפונקציה את כתובת המבנה ולהשתמש במצביעים.

פונקציות שמשנות ערכי שדות ברשומה

בפרקים הקודמים למדנו ששינוי ערך הפרמטרים על ידי פונקציה דורש העברת כתובת המשתנה אל הפונקציה. לכן, יש להעביר את כתובת משתנה הרשומה אל הפונקציה, כדי שהיא תוכל לבצע שינויים בערכי שדות הרשומה. העברת כתובת הרשומה לפונקציה נעשית על ידי הצבת אופרטור הכתובת (״כתובת של...״ &) לפני שם משתנה הרשומה. להלן דוגמה:

```
some_function(&worker);
```

בתוך הפונקציה יש לפנות לשדות הרשומה באמצעות מצביעים. להלן תחביר מומלץ של משפט לשימוש במצביעים לשדות הרשומה:

```
pointer_variable->member = some_value;
```

לדוגמה, בתוכנית **CHG_MBR.CPP** מעבירים רשומה מטיפוס employee לפונקציה get_employee_id. תפקיד הפונקציה הוא לבקש מהמשתמש להזין את מספר הזיהוי של העובד ולאחר מכן לאחסן אותו בשדה employee_id של הרשומה. כדי לשנות את ערך השדה, הפונקציה מפעילה מצביע לרשומה.

```
#include <iostream.h>
#include <string.h>

struct employee {
  char name[64];
  long employee_id;
  float salary;
  char phone[10];
  int office_number;
 };

void get_employee_id(employee *worker)
 {
   cout << "Type in an employee id: ";
   cin >> worker->employee_id;
 }

void main(void)
 {
   employee worker;

   // Copy a name to the string
   strcpy(worker.name, "John Doe");
   get_employee_id(&worker);

   cout << "Employee: " << worker.name << endl;
   cout << "Id: " << worker.employee_id << endl;
 }
```

בתוכנית הראשית main מעבירים לפונקציה get_employee_id את כתובת משתנה המבנה worker. בתוך הפונקציה get_employee_id מאחסנים בשדה employee_id את הערך שהמשתמש הזין. כותבים לצורך זה את המשפט הבא:

```
cin >> worker->employee_id;
```

שימוש במצביעים לרשומה

כאשר צריך לשנות ערכי שדות של הרשומה בתוך הפונקציה, יש להעביר לפונקציה את כתובת משתנה הרשומה. במקביל הפונקציה מקבלת את המשתנה על ידי שימוש במצביע לרשומה. כדי לגשת לשדה ברשומה יש להשתמש במצביעים על פי הפירוט הבא:

```
value = variable->member;
variable->other_member = some_value;
```

סיכום

לפני שנעבור לפרק הבא נבחן אם מובנים לנו הנושאים הבאים:

- ✓ מבנה הרשומה אוגר מידע שמורכב מנתונים מטיפוסים שונים.
- ✓ היחידות המרכיבות את הרשומה נקראות שדות (fields או member).
- ✓ מבנה הרשומה מגדיר תבנית (template) לצורך הגדרה עתידית של משתנים.
- ✓ לאחר הגדרת מבנה הרשומה ניתן להשתמש בשם המבנה (tag) להגדרת משתנים מאותו טיפוס.
- ✓ גישה לשדות הרשומה, או הקצאת ערכים לשדות, נעשית על ידי שימוש באופרטור הנקודה באופן הבא: variable.member (שדה.משתנה).
- ✓ כדי לשנות בתוך הפונקציה ערך של שדה ברשומה, יש להעביר אל הפונקציה את הכתובת של משתנה הרשומה כפרמטר.
- ✓ כדי לגשת לשדה ברשומה כאשר בפונקציה יש שימוש במצביע לרשומה, צריך להשתמש בפורמט: variable ->member.

בפרק 19 נציג כיצד משתמשים במבנה **האיגוד** (**union**) בשפת ++C. מבנה האיגוד דומה למבנה הרשומה, מכיון שהוא מורכב משדות. לעומת זאת, אחסון מבנה האיגוד בזיכרון אינו זהה לאחסון מבנה הרשומה. מבנה האיגוד מסוגל לאחסן בתוכו ערך אחד ברגע נתון, ללא קשר למספר השדות שבו.

תרגילים

1. הגדירו רשומה הכוללת את הפרטים הבאים לגבי גבר/אישה: שם, גיל. יש להגדיר משתנה מטיפוס הרשומה. יש לאתחל את שדות הרשומה מהקלט (cin) ולאחר מכן להדפיס את הפרטים למסך.
2. כיתבו תוכנית המנהלת יומן פגישות. הגדירו רשומה המכילה שדות אלה: יום הפגישה, חודש הפגישה ומחרוזת המתארת את הפגישה. יש ליצור מערך של 20 רשומות (היומן) ולכתוב שיגרה המאפשרת למשתמש להזין פגישה חדשה ליומן, ושיגרה נוספת המדפיסה את רשימת הפגישות.

3. כיתבו תוכנית המגדירה רשומה המכילה נתונים לגבי אורך שתי הצלעות המאונכות של משולש ישר זווית. כיתבו פרוצדורה המקבלת כפרמטר רשומה מסוג זה, ומציירת את המשולש בעזרת כוכביות ('*'). יש לכתוב שיגרת main אשר תקלוט את נתוני המשולש ותדפיס את ציור המשולש.

4. נתון מבנה הרשומה הבא:

```
struct grade {
   char name[20];
   float exam_grade[4];
   int passed;
}
```

כיתבו שיגרה המקבלת משתנה מטיפוס grade ומעדכנת בו את ערך המשתנה passed, כך שיקבל ערך 1 אם ממוצע הציונים גדול מ-60 ואחרת - יקבל ערך 0.

פרק 19

איגודים

בפרק 18 למדנו כיצד מאחסנים מידע בתוך מבנה רשומה. במקרים מסוימים יש צורך שהתוכנית תתייחס למידע בדרכים שונות. בנוסף, לעיתים התוכניות צריכות לעבד מספר ערכים, אך ברגע נתון עליהן להתייחס לערך מסוים אחד בלבד. הפתרון לשני המקרים הקודמים הוא אגירת המידע במבנה **איגוד** (**Union**).

בהמשך הפרק נדון בנושאים אלה:

- **איגודים** (Unions) בשפת ++C דומים מאוד למבני רשומה, אך נבדלים מהם בצורה בה ++C מאחסנת כל אחד מהם בזיכרון, וגם בכך שאיגוד יכול לאחסן בכל זמן נתון ערך של שדה אחד בלבד.
- איגוד הוא מבנה נתונים, אשר כמו מבנה רשומה מכיל קטעי נתונים, הנקראים שדות.
- איגוד מגדיר **תבנית** (Template), אשר באמצעותה ניתן להצהיר על משתנים.
- גישה לשדה של איגוד נעשית באמצעות **אופרטור הנקודה** (**.**) של ++C.
- כדי לשנות ערך של שדה של איגוד בתוך פונקציה, צריך להעביר אל הפונקציה את כתובת משתנה האיגוד.
- **איגוד אנונימי** (Anonymous) הוא איגוד ללא שם.

בהמשך נראה שמבנה האיגוד דומה מאוד למבנה הרשומה, בשפת ++C, אך מאחסנים אותו באופן שונה מאופן אחסון של מבנה הרשימה.

אחסון מבנה האיגוד

מבחינת הגדרתם בתוכנית, מבנה האיגוד ומבנה הרשומה דומים מאוד. למשל, המשפטים הבאים מגדירים מבנה איגוד בשם distance, המורכב משני שדות.

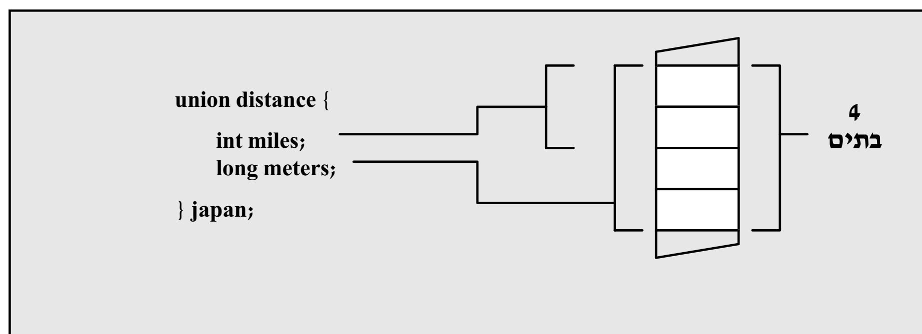
```
union distance {
  int miles;
  long meters;
};
```

בדומה למבנה הרשומה, הגדרת מבנה האיגוד אינה מאוחסנת בזיכרון המחשב. כלומר, אין עבורה הקצאת זיכרון. הגדרה זו היא תבנית (Template) למשתנים מטיפוס האיגוד, אשר יוגדרו בשלב מאוחר יותר בתוכנית. כדי להגדיר משתנים מטיפוס איגוד, אפשר להשתמש באחד משני הפרמטרים הבאים:

```
union distance {
  int miles;
  long meters;
};
distance japan, germany,france;
```

```
union distance {
  int miles;
  long meters;
} japan, germany, france;
```

מבנה האיגוד שבדוגמה מכיל שני שדות: miles ו- meters. ההגדרות יוצרות משתנים המיועדים לאחסון המרחק למדינות ספציפיות. בדומה למבנה הרשומה, ניתן להציב ערך לשדות האיגוד, אך בניגוד לו, ניתן לתת ערך לשדה אחד בלבד ברגע נתון. בעת הגדרת האיגוד, המהדר מקצה מקום בזיכרון לאחסון השדה הגדול ביותר המוגדר בתבנית האיגוד. במקרה של איגוד distance, המהדר יקצה בזיכרון מקום לאחסון ערך מטיפוס long. תרשים 19.1 מתאר את הקצאת הזיכרון לאיגוד distance.



תרשים 19.1: מהדר C++ מקצה בזיכרון מקום לשדה הגדול ביותר באיגוד.

נניח כי בתוכנית מתבצעת השמה של ערך לשדה miles, לפי המשפט הבא:

```
japan.miles = 12123;
```

כאשר בהמשך תתבצע השמה לשדה meters, הערך החדש ידרוס את הערך הקיים בשדה miles.

בתוכנית **USEUNION.CPP** מתואר יישום הנעזר בהגדרת האיגוד distance. בתחילת התוכנית מקצים ערך לשדה miles ומציגים אותו על המסך. בהמשך מקצים ערך לשדה meters. בשלב זה יידרס (יימחק) ערך השדה miles על ידי הערך החדש.

```
#include <iostream.h>

void main(void)
 {
  union distance {
   int miles;
   long meters;
  } walk;

  walk.miles = 5;

  cout << "A distance walked in miles is " << walk.miles << endl;

  walk.meters = 10000;

  cout << "A distance walked in meters is " << walk.meters << endl;
 }
```

יש לשים לב שהגישה לשדות האיגוד מתבצעת תוך שימוש באופרטור הנקודה. בצורה דומה מתאפשרת גישה גם לשדות הרשומה, כפי שמוצג בפרק 18.

איגוד מאחסן ערך של שדה אחד בלבד בכל זמן נתון

איגוד הוא מבנה נתונים (data structure), אשר כמו מבנה רשומה, מאפשר לקבץ בתוך משתנה אחד מספר נתונים מטיפוסים שונים, אך קשורים זה לזה. עם זאת, שלא כמו מבנה רשומה, איגוד מאחסן ערך של שדה אחד בלבד בכל זמן נתון. במילים אחרות, כאשר מקצים ערך לשדה כלשהו של איגוד, הוא "דורס" (מוחק, ובא במקום) את הערך הקודם אשר היה מאוחסן בשדה קודם לכן. האיגוד מגדיר תבנית (Template) שבאמצעותה ניתן להצהיר על משתנים בתוכנית. כאשר המהדר נתקל בהגדרת איגוד, הוא מקצה רק את כמות הזיכרון המאפשרת לו לאחסן ערכים עבור השדה הגדול ביותר של האיגוד, שהתוכנית עלולה לפגוש.

האיגוד האנונימי

האיגוד האנונימי (**Anonymous union**) הוא איגוד אשר אין לו שם מוגדר. שפת C++ מציעה סוג זה של איגוד כדי להקל על הגישה לשדות האיגודים המוגדרים בתוכניות, כאשר מטרת האיגוד הוא לחסוך בשימוש בזיכרון המחשב, או ליצור שמות נרדפים לערכים מסוימים. נניח, למשל כי בתוכנית מסוימת זקוקים לשינוי בשני משתנים miles ו-meters. בנוסף ידוע, כי התוכנית משתמשת באחד המשתנים ברגע נתון. לפני כן הצגנו את השימוש באיגוד distance שיכול לשמש פתרון לבעיה המוצגת כאן. במקרה זה נגדיר איגוד בשם name וניגש לשדות על ידי שימוש באופרטור הנקודה: name.miles ו-name.meters. המשפט הבא יוצר (מגדיר) איגוד אנונימי (ללא שם):

```
union {
  int miles;
  long meters;
};
```

בהגדרה הקודמת ניתן לראות שאין לאיגוד שם, ואין הגדרה של משתנה מטיפוס האיגוד. הגישה לשדות האיגוד miles ו-meters מתוך התוכנית נעשית ללא שימוש באופרטור הנקודה. בתוכנית הבאה **ANONYM.CPP** מוגדר איגוד אנונימי המכיל שני שדות: miles ו-meters. יש לשים לב שהגישה לשדות האיגוד מתבצעת בצורה ישירה בדומה לגישה אל משתנים רגילים. קיים הבדל מהותי בין השימוש במשתנים רגילים לבין האיגוד המתואר, מכיון שברגע שמציבים ערך לשדה אחד של האיגוד אין אפשרות להשתמש בערך של שדה אחר שלו.

```
#include <iostream.h>

void main(void)
 {
   union {
     int miles;
     long meters;
   };

   miles = 10000;

   cout << "The value of miles is " << miles<< endl;

   meters = 150000L;
   cout << "The value of meters is " << meters << endl;
 }
```

הדוגמה הקודמת מוכיחה שהשימוש באיגוד אנונימי מאפשר חיסכון בזיכרון המחשב, מבלי להשתמש בשם האיגוד ובאופרטור הנקודה, כדי לגשת לערכי השדות של האיגוד.

איגודים אנונימיים חוסכים מקום בזיכרון

איגוד אנונימי (Anonymous union) הוא איגוד חסר שם. איגודים אנונימיים מאפשרים לחסוך במקום, מבלי לטרוח עם שימוש באופרטור הנקודה (.). המשפטים הבאים מגדירים איגוד אנונימי המסוגל לאחסן שתי מחרוזות תווים :

```
union  {
    char short_name [13];
    chat long_name [255];
 };
```

סיכום

בפרק זה למדנו כיצד להגדיר **איגוד** (Union) וכיצד להשתמש בו בתוכנית. הצגנו שמבנה האיגוד דומה למבנה הרשומה. למרות זאת, קיים שוני מהותי בין שני המבנים בקשר לצורת אחסון המידע בזיכרון המחשב.

בפרק 10 למדנו שכדי שפונקציה תוכל לשנות פרמטר, התוכנית צריכה להעביר את הפרמטר אל הפונקציה באמצעות מצביע (או כתובת זיכרון). השתמשנו במצביעים לעבודה במערכים ובמחרוזות תווים. בפרק הבא נחזור ונעסוק בפעולות המצביעים בשפת ++C.

לפני שנעבור לפרק הבא נבחן אם מובנים לנו הנושאים הבאים :

- ✓ המהדר מקצה קטע זיכרון התואם לשדה הגדול ביותר, המוגדר באיגוד.
- ✓ הגדרת מבנה האיגוד איננה גורמת להקצאת זיכרון. היא מהווה תבנית מתאימה להגדרה עתידית של משתנים מטיפוס האיגוד.
- ✓ הגישה לשדות האיגוד מתבצעת על ידי שימוש באופרטור הנקודה. השמה של ערך בשדה כלשהו של האיגוד גורם למחיקה של הערך הקיים בשדה אחר של אותו איגוד.
- ✓ איגוד אנונימי הינו איגוד ללא שם. הגישה לשדות איגוד אנונימי בתוכנית זהה לגישה אל משתנים רגילים, ללא צורך באופרטור הנקודה.

תרגילים

1. מהו הפלט של התוכנית הבאה? כמה זיכרון מוקצה למבנה union?

```
void main(void)
 {
    union currency {
       int dollars;
       int new_shekalim;
       long yens;
     } amount;

    amount.dollars = 10000;
    cout << "I wish I had " << amount.dollars << "$"
         << endl;

    amount.new_shekalim = amount.dollars * 3;
    cout << "which means that I wish I had "
         << amount.new_shekalim << " NIS" << endl;

    amount.yens = (long) amount.new_shekalim * 40;
    cout << "which means that I wish I had "
         << amount.yens << " YENs" << endl;

    cout << "But I can only dream about heaving "
         << amount.yens << "$ ..." << endl;
 }
```

2. מהו הפלט של התוכנית הבאה?

```
#include <iostream.h>
void main(void)
  {
    union pair {
       int x;   int y;
     } p;
    p.x = 3;
    cout << p.x << endl;
    p.y = 5;
    cout << p.y << endl;
    cout << p.x << endl;
    p.x = 7;
    cout << p.y << endl;
    cout << p.x << endl;
  }
```

פרק 20

מצביעים

בפרקים הקודמים ראינו ששינוי ערכי הפרמטרים בתוך הפונקציה מחייב העברת כתובות הפרמטרים לפונקציה. הגישה לפרמטר בתוך הפונקציה נעשית על ידי שימוש ב**מצביעים (Pointers)**. במספר תוכניות שהוצגו בפרקים הקודמים הפעלנו את שיטת המצביעים לפרמטרים. בדרך כלל, בתוכניות המבוססות על מערכים או מחרוזות, משתמשים במצביעים לניהול וגישה לאיברי המערך.

בהמשך הפרק נדון בנושאים הבאים:

- יישום פעולות המצביעים לטיפול במחרוזות: מחרוזת תווים כמצביע, וטיפול בתוכן המחרוזת באמצעות פעולות במצביעים.
- פעולות חשבוניות במצביעים כמו הגדלת ערך המצביע באחד.
- שימוש במצביעים לטיפול במערכים מטיפוס int, או float.

פעולות המצביעים (Pointer operations) מקובלות בתוכניות ++C, ולכן מומלץ להשקיע זמן בתרגול והרצה חוזרת של התוכניות שבפרק זה.

מצביע למחרוזת תווים

המצביע מכיל ערך של **כתובת** בזיכרון המחשב. כאשר התוכנית מעבירה מערך (לדוגמה, מחרוזת תווים) כפרמטר לפונקציה, שפת C++ מעבירה את כתובת הזיכרון של האיבר הראשון. מסיבה זו נפוץ השימוש במצביעים למחרוזות. הגדרת מצביע למחרוזת נעשית על ידי הצבת כוכבית לפני שם המשתנה, כדוגמת המשפט המופיע להלן:

```
void some_function(char *string);
```

בתוכנית **PTR_STR.CPP** משתמשת במצביע למחרוזת בפונקציה show_string. תפקיד הפונקציה הוא להציג את תווי המחרוזת על המסך, תו אחר תו.

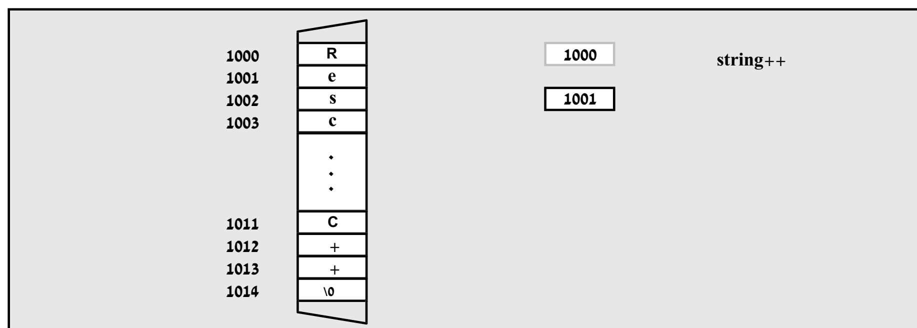
```
#include <iostream.h>

void show_string(char *string)
 {
   while (*string != '\0')
     {
       cout << *string;
       string++;
     }
 }

void main(void)
 {
   show_string("Rescued By C++!");
 }
```

כעת נשים לב ללולאה while שבתוך הפונקציה show_string. תנאי הלולאה while (*string != '\0') בודק אם התו שהמצביע string מורה עליו ברגע הבדיקה אינו התו NULL, אשר מציין את התו האחרון במחרוזת. אם התשובה חיובית והתו אינו NULL, הפקודה cout בלולאה גורמת להצגת התו על המסך. בהמשך, המשפט ++string מקדם את המצביע string, כדי שיצביע על האיבר הבא במחרוזת. כאשר המצביע string מצביע על התו NULL, תושלם הצגת המחרוזת, והלולאה תסתיים.

נניח למשל, שכתובת הזיכרון של המחרוזת המועברת לפונקציה היא 1000. כל פעם שערך המצביע string יתקדם באחד, המצביע string יצביע על התו הבא במחרוזת (כתובות 1001, 1002, 1003 וכו׳). דוגמה זו מתוארת בתרשים 20.1.



תרשים 20.1: סריקת מחרוזת באמצעות מצביע.

דוגמה נוספת

זה עתה למדנו שעל ידי שימוש במצביע ניתן לסרוק מחרוזות עד לתו NULL המציין את סוף המחרוזת. בתוכנית **PTR_LEN.CPP** מוגדרת פונקציה string_length המנצלת את תכונות המצביע של המחרוזת, כדי לקבוע את גודל המחרוזת.

```
#include <iostream.h>

int string_length(char *string)
 {
   int length = 0;

   while (*string != '\0')
    {
      length++;
      string++;
    }

   return(length);
 }

void main(void)
 {
   char title[] = "Rescued By C++";

   cout << title << " contains " <<
                  string_length(title) << " characters";
 }
```

בפונקציית string_length הלולאה מתבצעת עד לרגע שמצביע מחרוזות מגיע לתו NULL.

קידום מצביע למחרוזת תווים

כאשר תוכנית מעבירה מערך אל פונקציה, היא למעשה מעבירה לה את כתובת הזיכרון של האיבר הראשון במערך. השימוש במצביע מאפשר לפונקציה לנוע לאורך המערך על ידי קידום ערך המצביע. לדוגמה, נניח כי התוכנית מעבירה לפונקציה מסוימת את מחרוזת התווים "Hello". תחילה פונה המצביע אל המקום בזיכרון (כתובת) שבו נמצא התו 'H'. כאשר הפונקציה מקדמת את המצביע, הוא מצביע כעת אל הכתובת בזיכרון שבה נמצא התו 'e'. הפונקציה ממשיכה לקדם את המצביע אל כתובות הזיכרון שבהן מאוחסנים שאר איברי המערך, עד אשר המצביע מגיע אל התו NULL.

צמצום משפטים עודפים

בשתי הדוגמאות הקודמות השתמשנו במשפט לולאת while הבא, כדי לקבוע את סוף המחרוזת:

```
while (*string != '\0')
```

כידוע, ערך התו NULL ('\0') בקוד ASCII הוא אפס. בשפת C++ משתמשים בערך 0 כדי לציין ערך שקרי (false), ובכל ערך אחר, כדי לציין ערך "אמת" (true), ראה בפרק 7. על כן, ניתן לכתוב את משפט while הקודם גם בצורה זו:

```
while (*string)
```

במקרה זה, כל עוד המצביע אינו מגיע לערך 0 (NULL), תנאי הבדיקה מתקיים והלולאה נמשכת.

בפרק 5 למדנו את תכונותיו של אופרטור ההגדלה המאוחרת בשפת C++ (Postfix increment operator), שבעזרתו משתמשים בערך המאוחסן במשתנה, ואחר כך מקדמים אותו באחד. בתוכניות רבות הכתובות בשפת C++ משתמשים באופרטור זה ובאופרטור ההקטנה המאוחרת (Postfix decrement operator) לשם סריקת מערכים על ידי מצביעים. לדוגמה, שתי הלולאות הבאות פועלות בצורה זהה לחלוטין:

```
while (*string)
cout << *string++;
```

```
while (*string)
 {
       cout << *string;
       string++;
 }
```

המשפט "cout << *string++" גורם להצגת תו המחרוזת המוצבע על ידי המצביע string ולהגדלת ערך המצביע, כך שיצביע על התו הבא במחרוזת. טכניקה זו מיושמת בתוכנית **SMARTPTR.CPP**. בתוכנית מוגדרות מחדש הפונקציות show_string ו-string_length.

```
#include <iostream.h>

void show_string(char *string)
 {
   while (*string)
     cout << *string++;
 }

int string_length(char *string)
 {
   int length = 0;

   while (*string++)
     length++;

   return(length);
 }

void main(void)
 {
   char title[] = "Rescued By C++";

   show_string(title);
   cout << " contains " <<
     string_length(title) << " characters";
 }
```

יש לשים לב, שהשימוש בטכניקת המצביעים המקוצרת, המתוארת בתוכנית, יעיל מאוד כאשר הפונקציה מטפלת במחרוזות.

סריקת מחרוזת תווים

אחד השימושים השכיחים ביותר של מצביעים הוא סריקת מחרוזות תווים. על מנת לחסוך בעבודת קידוד תוכניות רבות משתמשות באוסף משפטים מהצורה הבאה כדי לסרוק מחרוזת:

```
while  (*string)
  {
       // statements
      string++;     // point to the next character
  }
```

הפונקציה הבאה, string_uppercase, משתמשת במצביעים כדי להמיר את תווי המחרוזת לאותיות גדולות (Uppercase):

```
char *string_uppercase (char string)
 {
  char *starting_address = string; // address of string[0];
  while  (*string)
     {
       if ((*string >= 'a')  &&  (*string <= 'z'))
         *string = *string - 'a' + 'A';
       string++;
     }
   return (starting_address);
 }
```

הפונקציה מחזירה את הכתובת ההתחלתית של המחרוזת המעודכנת, ולכן ניתן להשתמש בפונקציה באופן הבא:

```
cout << string_uppercase ("Hello, world!") << endl;
```

הפעלת מצביעים במערכים מטיפוסים שונים

השימוש במצביעים נפוץ לטיפול במחרוזות, אולם ניתן לנצל את תכונות המצביע לטיפול במערכים מטיפוסים אחרים. לדוגמה, בתוכנית **PTRFLOAT.CPP** מתואר השימוש במצביע למערך מטיפוס float.

```
#include <iostream.h>

void show_float(float *array,
  int number_of_elements)
 {
   int i;
   for (i = 0; i < number_of_elements; i++)
     cout << *array++ << endl;
 }

void main(void)
 {
   float values[5] = {1.1, 2.2, 3.3, 4.4, 5.5};

   show_float(values, 5);
 }
```

הפונקציה show_flow המוגדרת בתוכנית, נעזרת בלולאה for להצגת ערך האיבר שהמצביע array מורה עליו. לאחר מכן, הלולאה מקדמת את ערך המצביע array כדי שיצביע על האיבר הבא. במקרה זה התוכנית מעבירה לפונקציה פרמטר נוסף, המציין את מספר האיברים שבמערך. ערך זה דרוש, מכיון שמערכים מטיפוס float או int, long וכו' אינם כוללים את התו המיוחד NULL לציון סוף המערך, כפי שקיים במחרוזות.

מתמטיקה של מצביעים

כפי שלמדנו, תוכניות יכולות להשתמש במצביעים למערך מכל טיפוס. בתוכנית הקודמת, ראינו כיצד הפונקציה show_float מקדמת מצביע בתוך מערך מטיפוס float. ניתן לומר שהמצביע מצביע אל מקום בזיכרון (כתובת), שבה מאוחסן ערך מטיפוס מוגדר, למשל, char, int או float. כאשר פונקציה סורקת מערך באמצעות מצביע, היא מקדמת את המצביע בכל פעם, מהאיבר הנוכחי אל האיבר שבא אחריו. כדי שמצביע יוכל להצביע לאיבר הבא במערך, על ++C לעקוב אחר גודל איברי המערך (בבתים) כדי לדעת בכמה בתים צריך לקדם את המצביע כל פעם. לדוגמה, כדי לקדם את המצביע אל התו הבא במחרוזת, ++C צריכה לקדם את ערך המצביע בבית אחד. לעומת זאת, כדי להצביע לאיבר הבא במערך מטיפוס int, על ++C לקדם את המצביע בשני בתים (ערכים מטיפוס int תופסים שני בתי זיכרון). עבור ערכים מטיפוס float, ++C מקדמת את המצביע בארבעה בתים. על ידי ידיעת טיפוס הערך שאליו מצביע המצביע, ++C יודעת בכמה בתים לקדם את ערך המצביע (pointer value), כך שיצביע אל האיבר הבא. בתוך התוכנית יש להשתמש באופרטור ההגדלה העצמית, כך למשל ++value. מאחורי הקלעים ++C מגדילה את ערכו של המצביע (את כתובת הזיכרון) במספר הבתים הדרוש.

סיכום

בתוכניות ++C נפוץ השימוש במצביעים, במיוחד לטיפול במחרוזות תווים.

לפני שנעבור לפרק הבא נבחן אם מובנים לנו הנושאים הבאים:

- ✓ המצביע מכיל ערך שהוא כתובת בזיכרון המחשב. כאשר התוכנית מעבירה מערך לפונקציה, היא מעבירה למעשה את הכתובת של האיבר הראשון במערך.
- ✓ הגדלת ערך המצביע באחד גורמת לכך שהמצביע יצביע על האיבר הבא של המערך.
- ✓ פונקציות המטפלות במחרוזות באמצעות מצביעים, סורקות בדרך כלל את המחרוזת עד שהמצביע מגיע לתו NULL (שבסוף המחרוזת).
- ✓ פונקציות המטפלות במערכים שאינם מחרוזות, צריכות לקבל את מספר האיברים של המערך המועבר, מכיון שלמערכים אלה אין ערך סיום מיוחד.

בפרק הבא נתחיל בהצגת התכונות של ++C כשפה **מונחית-עצמים** (**Object Oriented**). תחילה נציג את **המחלקה** (**Class**) אשר דומה ל**מבנה הרשומה** (**Structure**). המחלקה מאפשרת להגדיר עצמים. בהמשך נשתמש במחלקה כדי להגדיר את העצם **file** (**קובץ**). בתוך המחלקה המגדירה את העצם הזה נכללות הפונקציות המטפלות בו, וביניהן print_file ו- delete_file.

תרגילים

1. כיתבו שיגרה המקבלת מחרוזת והופכת בה את כל האותיות הלועזיות הקטנות לאותיות לועזיות גדולות. יש להשתמש במצביעים כדי לסרוק את תווי המחרוזת.

   יש לחפש בספריה <ctype.h> פונקציה מתאימה לבדיקת גודל האותיות.

2. כיתבו את השיגרה הבאה:

```
int str_in_str(char * str1, char * str2)
```

השיגרה תחזיר 1 אם המחרוזת str1 מכילה את המחרוזת str2, ואחרת - תחזיר 0.

3. כיתבו שיגרה המקבלת מחרוזת ומזהה אם זהו פלינדרום. כלומר, היא מחזירה 1 כאשר המחרוזת היא פלינדרום, ואחרת, היא מחזירה 0. לידיעתך, מחרוזת הינה פלינדרום, אם היא זהה ללא תלות בכיוון הקריאה, כשנתעלם מהרווחים. הנה מספר דוגמאות לפלינדרומים:

```
abba
racecar
step on no pets
names reverse man
```

יש להשתמש במצביעים לסריקת המחרוזת, וצריך להתעלם מתווי רווח שנמצאים בה.

4. כיתבו את השיגרה הבאה:

```
char* str_char(char* s, char c)
```

המחזירה מצביע אל מיקום תו "c" הראשון במחרוזת s. יש להחזיר NULL במקרה ש-c אינו מופיע ב-s.

חלק 4

המחלקות של C++

Object-Oriented Programming - תכנות מונחה עצמים מתמקד **בעצמים** או **דברים** - (**objects** או **things**) המרכיבים את מערכת התוכנה. לדוגמה, ניתן להגדיר עצם מטיפוס קובץ או, עצם מטיפוס עובד, או עצם מטיפוס אחר כלשהו. עצם מטיפוס עובד יכול להכיל נתונים, כגון, שם העובד, מספר זיהוי ושכר. עד כאן העצם דומה למבנה הרשומה, אולם העצם מורכב גם מסדרה של פעולות שהתוכנית מבצעת על הנתונים הכלולים בו. לדוגמה, בעצם מטיפוס קובץ נמנות פעולות רגילות כגון הדפסה, מחיקה או העתקת קובץ. לעצם מטיפוס עובד ניתן לייחס פעולות של הדפסה, קידום או **חיסול** של העצם. בשפת C++ המחלקה היא המעטפת של העצם אשר מכילה בתוכה את הנתונים והפונקציות שפועלות עליהם. בחלק זה של הספר נציג את המחלקה בצורה מפורטת.

חלק זה כולל את הפרקים הבאים:

פרק 21

המחלקות: תחילת הדרך

המחלקה (Class) היא כלי העבודה העיקרי בשפת ++C כשפת תכנות מונחה עצמים. בפרק הנוכחי נראה שהמחלקה דומה במידה רבה ל**מבנה הרשומה** (Structure). המחלקה היא המעטפת של המרכיבים של העצם, הכוללים את הנתונים ואת הפונקציות הפועלות עליו, בכך שהיא מקבצת נתונים או משתנים שמתייחסים לפריטי נתונים (**עצמים**, **אובייקטים**), או לפונקציות שפועלות על הנתונים האלה (שיטות, methods). שפת ++C מאפשרת לך להגדיר את כל המאפיינים (תכונות) של עצם. במקרה של נתוני עצם מוחשי מסוג טלפון, למשל, המחלקה תכיל שדות נתונים שיפרטו אם זה טלפון חוגה או טלפון לחצנים והפעולות שניתן לעשות בו. על ידי הקבצה של נתוני העצם למשתנה אחד (המבנה), אפשר לפשט את התכנות ואת אפשרות השימוש החוזר בקוד.

בהמשך הפרק נדון בנושאים הבאים :

- להגדרת המחלקה בתוכנית צריך לפרט את המשתנים ואת הפונקציות הכלולים בה.
- הגדרת המחלקה כוללת תבנית (Template) שבאמצעותה התוכנית יכולה ליצור עצמים מטיפוס המחלקה, כשם שתוכנית יוצרת משתנים מטיפוס int, char וכד'.
- השמה של ערכים לנתונים של המחלקה באמצעות אופרטור הנקודה.
- קריאה לפונקציות המחלקה באמצעות אופרטור הנקודה.

העצם ותכנות מונחה עצמים

במובן הפשוט ביותר **עצם** (Object - אובייקט) הוא **דבר** (Thing). בתוכניות הרגילות, המשתנים מוגדרים לאחסון נתונים אודות דברים של העולם האמיתי, כמו לדוגמה עובדים, ספרים וגם קבצים. עיצוב תוכנית המתבסס על שיטת תכנות מונחה עצמים מתמקד ב**דברים** (Things) המרכיבים את מערכת התוכנה (היישום) ואת הפעולות שיש לבצע על אותם הדברים. לדוגמה, הפעולות שיש לביצוע בעצם מטיפוס קובץ הן הדפסה, הצגה, מחיקה ועדכון. בשפת ++C המחלקה היא התבנית להגדרת העצם. מטרת התכנות היא להכניס בתוך המחלקה את המידע המירבי אודות העצם. בדרך זו ניתן לנצל את הגדרת המחלקה במספר רב של תוכניות.

המחלקה מורכבת מאלמנטים שהם נתונים ופונקציות הפועלות על הנתונים האלה. בספרות המקצועית הדנה בתכנות מונחה עצמים מתייחסים לפונקציות הפועלות על נתוני המחלקה במונח **שיטות** (**Methods**). בדומה למבנה הרשומה, המחלקה המוגדרת בתוכנית חייבת להיות בעלת שם ייחודי בה. במשפט הגדרת המחלקה, מרכיביה נרשמים בתוך סוגריים מסולסלים.

```
class class_name {
  int data_member;              // Data member
  void show_member(int);        // Function member
};
```

לאחר הגדרת המחלקה ניתן להגדיר משתנים שהם העצמים (Objects) מטיפוס המחלקה הזו. להלן דוגמה של הגדרת עצמים:

```
class_name object_one, object_two, object_three;
```

בהמשך מוצגת הגדרה של מחלקה בשם employee. המחלקה מכילה משתנים לאחסון הנתונים ופונקציות או שיטות הפועלות עליהם.

```
class employee {
 public:
  char name[64];
  long employee_id;
  float salary;
  void show_employee(void)
   {
     cout << "Name: " << name << endl;
     cout << "Id: " << employee_id << endl;
     cout << "Salary: " << salary << endl;
   };
};
```

המחלקה שבדוגמה מורכבת משלושה משתנים ופונקציה אחת. יש לשים לב לתווית "public:" המופיעה בתוך הגדרת המחלקה. בפרק 22 נלמד שאלמנט במחלקה יכול להיות **פרטי** (**Private**), או **ציבורי** (**Public**). דבר זה קובע כיצד התוכנית יכולה לפנות לאלמנט. במקרה הקודם, כל האלמנטים נמצאים בחלק הציבורי של המחלקה, ולכן התוכנית יכולה לגשת אליהם על ידי שימוש באופרטור **הנקודה** (**Dot operator**) של שפת ++C. לאחר הגדרת המחלקה בתוכנית ניתן להגדיר עצמים (משתנים) מטיפוס המחלקה. כפי שנראה בדוגמה הבאה:

שם המחלקה

```
employee worker, boss, secretary;
```

משתני המחלקה (עצמים)

בתוכנית **EMPCLASS.CPP** מוגדרים שני עצמים מטיפוס המחלקה employee. השימוש באופרטור הנקודה מאפשר לתוכנית להציב ערכים למשתני המחלקה. בהמשך התוכנית מופעלת פונקציית המחלקה show_employee, כדי להציג את הנתונים של העובדים.

מומלץ להגדיר את המחלקה בקובץ כותר h, ולהכליל אותה בקובץ cpp כ-#include.

```
#include <iostream.h>

class employee {
 public:
  char name[64];
  long employee_id;
  float salary;
  void show_employee(void)
   {
     cout << "Name: " << name << endl;
     cout << "Id: " << employee_id << endl;
     cout << "Salary: " << salary << endl;
   };
};
```

```
#include <iostream.h>
#include <string.h>
#include "empclass.h"

void main(void)
 {
   employee worker, boss;
```

```
  strcpy(worker.name, "John Doe");
  worker.employee_id = 12345;
  worker.salary = 25000;

  strcpy(boss.name, "Happy Jamsa");
  boss.employee_id = 101;
  boss.salary = 101101.00;

  worker.show_employee();
  boss.show_employee();
}
```

בתוכנית מוגדרים שני עצמים מטיפוס employee, האחד בשם worker והשני בשם boss. בהמשך מתבצעת השמה של נתונים למשתני העצמים תוך שימוש באופרטור הנקודה. בסוף התוכנית קוראים לפונקציה show_employee ומפעילים אותה.

מהם עצמים (אובייקטים)?

ברוב תוכניות ++C נמצא התייחסות לישויות, או עצמים, שבעולם הממשי. לפי הפירוש הפשטני ביותר, עצם (Object, אובייקט) הוא דבר ממשי, כמו למשל מכונית, כלב או שעון. לעצם יש מספר תכונות וגם פעולות שהתוכנית אמורה לבצע על אותן תכונות. לדוגמה, במקרה של העצם מטיפוס שעון, התכונות (שדות הנתונים) עשויות לכלול את השעה הנוכחית, ואת השעה שנקבעה לצלצול המעורר. הפעולות שהתוכנית תבצע על השעון תהיינה כיוון השעה, כיוון השעון המעורר או כיבויו. התכנות מונחה העצמים (Object-oriented programming) מביא להתמקדות בעצמים שעליהם ובאמצעותם פועלת התוכנית, ובפעולות שניתן לבצע עליהם.

הגדרת פונקציית מחלקה מחוץ להגדרת מחלקה

במחלקה employee הקודמת מוגדרה פונקציית המחלקה בתוך המחלקה במקרה זה הפונקציה נקראת **פונקציה משולבת** (**Inline function**). הגדרת הפונקציה עלולה להקשות על הבנת המחלקה עצמה, כאשר הגדרת הפונקציה מורכבת יותר והקוד שלה מתרחב. במקום להציב את הגדרת הפונקציה במלואה בתוך הגדרת המחלקה, ניתן להגדיר את הפונקציה מחוץ למחלקה, ולהכניס בתוך המחלקה רק את **הצהרת הפונקציה** (**Function declaration**), אב-טיפוס (**Prototype**).

הגדרת המחלקה תיראה כמו בדוגמה הבאה:

```
class employee {
 public:
  char name[64];
  long employee_id;
  float salary;
  void show_employee(void);
};
```

אב טיפוס של פונקציה

פונקציות שונות עשויות לשאת את אותו שם ועל כן, כאשר מגדירים פונקציה מחוץ למחלקה, יש להציב את שם המחלקה ואופרטור טווח הכרה (::) לפני שם הפונקציה. להלן ההגדרה החדשה של הפונקציה show_employee.

שם המחלקה

שם הפונקציה

```
void employee::show_employee(void)
 {
   cout << "Name: " << name << endl;
   cout << "Id: " << employee_id << endl;
   cout << "Salary: " << salary << endl;
 };
```

בתוכנית **CLASSFUN.CPP** מוצגת הגדרת הפונקציה show_employee מחוץ למחלקה, ולשם כך משתמשים באופרטור הגלובלי כדי לציין את שם המחלקה.

הערה



מומלץ לבנות את המחלקה (class) בקובץ כותר h, את הגדרות הפונקציה השייכות למחלקה - בקובץ cpp, ואת הקובץ הראשי לבנות בקובץ cpp נפרד.

cpp	cpp	h
הקובץ הראשי main	פונקציות המחלקה	המחלקה class

יש ליצור פרויקט ולהריץ את התוכנית. הנחיות לבניית פרויקט, ראה בפרק 41 "יצירת פרויקט".

```
#include <iostream.h>
#include <string.h>

class employee {
 public:
  char name[64];
  long employee_id;
  float salary;
  void show_employee(void);
};
```

```
void employee::show_employee(void)
 {
   cout << "Name: " << name << endl;
   cout << "Id: " << employee_id << endl;
   cout << "Salary: " << salary << endl;
 };

void main(void)
 {
   employee worker, boss;

   strcpy(worker.name, "John Doe");
   worker.employee_id = 12345;
   worker.salary = 25000;

   strcpy(boss.name, "Happy Jamsa");
   boss.employee_id = 101;
   boss.salary = 101101.00;

   worker.show_employee();
   boss.show_employee();
 }
```

שיטות (פונקציות) המחלקה

השימוש במחלקות מאפשר לקבץ את נתוני העצם ואת הפונקציות (שיטות) אשר פועלות על נתונים אלה, בתוך משתנה אחד (העצם, האובייקט). כאשר מגדירים שיטה (Method) של עצם, ניתן לעשות זאת באחת משתי הדרכים הבאות: לכלול את קוד הפונקציה בתוך הגדרת המחלקה, או לכתוב את משפטי הפונקציה מחוץ למחלקה. המקרה הראשון, שבו כוללים את השיטה/הפונקציה בתוך המחלקה, הוא לכאורה נוח יותר, אך במהרה תיווכח שהגדרה כזו הינה מסורבלת ומקשה על המשתמש במחלקה לזהות את השירותים שהיא מספקת. במקרה השני, שבו הפונקציה מוגדרת מחוץ למחלקה, יש לציין בתוך המחלקה את אב הטיפוס של כל אחת מהשיטות/הפונקציות (כלומר, שם הפונקציה, הערך המוחזר על ידה, מספר הפרמטרים שהיא מקבלת והטיפוס שלהם), את ההכרזה על הפונקציה.

כדי להגדיר פונקציית מחלקה מחוץ להגדרת המחלקה, יש לכתוב לפני שם הפונקציה את שם המחלקה ואת אופרטור טווח ההכרה:

```
return_type  class_name::function_name(parameters)
  {
    // Statements
  }
```

דוגמה נוספת

בתוכנית **PEDIGREE.CPP** מוגדרת המחלקה dogs, אשר מכילה מספר שדות נתונים ואת הפונקציה show_breed. כפי שהצגנו קודם, הגדרת פונקציית המחלקה נמצאת מחוץ למחלקה עצמה. התוכנית יוצרת שני עצמים מטיפוס dogs, ובמהלך הרצתה היא מציגה על המסך את המידע המתייחס לכל אחד מהם :

```
#include <iostream.h>
#include <string.h>

class dogs {
 public:
  char breed[64];
  int average_weight;
  int average_height;
  void show_breed(void);
};

void dogs::show_breed(void)
 {
   cout << "Breed: " << breed << endl;
   cout << "Average Weight: " << average_weight << endl;
   cout << "Average Height: " << average_height << endl;
 }

void main(void)
 {
   dogs happy, matt;

   strcpy(happy.breed, "Dalmatian");
   happy.average_weight = 58;
   happy.average_height = 24;

   strcpy(matt.breed, "Shetland Sheepdog");
   matt.average_weight = 22;
   matt.average_height = 15;

   happy.show_breed();
   matt.show_breed();
 }
```

סיכום

בתוכניות הכתובות בשפת ++C נוהגים להשתמש במחלקות. כפי שהצגנו, הגדרת המחלקה דומה להגדרת מבנה הרשומה. בפרקים הבאים נציג את התכונות, העוצמה ואת היכולת של המחלקה.

לפני שנעבור לפרק הבא, נבדוק אם מובנים הנושאים הבאים:

- ✓ במובן הפשוט, העצם הוא הדבר אשר עליו התוכנית מבצעת את פעולותיה.
- ✓ תוכניות בשפת ++C מציגות עצמים באמצעות המחלקה.
- ✓ המחלקה, בדומה למבנה הרשומה, מכילה אלמנטים. חלק מהאלמנטים מיועדים לאחסון מידע (נתונים) וחלק מיועד לבצע פעולות על המידע (פונקציות/שיטות).
- ✓ לכל מחלקה בתוכנית חייב להיות שם ייחודי.
- ✓ לאחר הגדרת המחלקה ניתן להגדיר עצמים מטיפוס המחלקה המוגדרת.
- ✓ הגישה לאלמנטים של המחלקה מתאפשרת על ידי השימוש באופרטור הנקודה.
- ✓ ניתן להגדיר את פונקציות המחלקה בתוך הגדרת המחלקה, או מחוצה לה. כאשר מגדירים פונקציה מחוץ להגדרת המחלקה, יש להציב לפני שם הפונקציה את שם המחלקה ואת אופרטור טווח ההכרה לפי הפורמט class::function.

בפרק זה התייחסנו לתווית **:public** המופיעה בתוך הגדרת המחלקה. הסברנו שהצבת המילה בתוך ההגדרה מאפשרת גישה לכל האלמנטים של המחלקה מכל מקום בתוכנית. בפרק 22 נתאר בצורה מפורטת את חלקי המחלקה: החלק הפרטי והציבורי.

תרגילים

1. כיתבו מחלקה המשמשת כמונה (counter) ובעלת הממשקים הבאים: initialize - אתחול המונה לערך הנתון, inc ו-dec לעדכון ערך המונה (הוספה וגריעה, בהתאמה), ו-go המדפיסה את המספרים מ- 0 ועד ערך המונה.
2. כיתבו מחלקה המתארת צפרדע. נתוני המחלקה הם: name, color, age, weight. פונקציות המחלקה הן: print_status - המדפיסה את כל הנתונים של הצפרדע; quack - המדפיסה "!!!quack" ומצפצפת בעזרת פעמון המחשב, age*weight)/2)] פעמים.
3. כיתבו מחלקה Person אשר מכילה שם של איש/אישה ומערך מצביעים (בגודל 5 לחמישה חברים שלו/שלה (אלה הם עצמים מטיפוס Person). המחלקה תספק את השרותים האלה:
   - ❖ set_name - קביעת שם איש/אישה.
   - ❖ set_frnd - הוספת חבר/ה לרשימת החברים (בצורה מעגלית, כך שהכניסה השישית באה על חשבון המקום של הכניסה הראשונה).
   - ❖ print_status - הדפסת רשימת החברים.
   - ❖ initialize (אופציונלית) - אתחול משתני המחלקה.

פרק 22

החלק הפרטי והחלק הציבורי

בפרק הקודם הגדרנו שתי מחלקות אשר בתוכן הצבנו את המילה public. כפי שהסברנו, המילה הזו מאפשרת גישה לכל האלמנטים של המחלקות המוגדרות על ידי שימוש באופרטור הנקודה. בפרק זה נסביר כיצד ניתן לשלוט על הגישה לאלמנטים של המחלקה על ידי הצבתם **בחלק הפרטי** (**Private**), או **בחלק הציבורי** (**Public**) של המחלקה.

בהמשך הפרק נדון בנושאים הבאים:

- הגדרת אלמנטים פרטיים ואלמנטים ציבוריים במחלקה.
- הבנת הסתרת מידע (Information hiding), וכיצד להשתמש בה בתוכנית.
- השימוש בפונקציות ממשק (Interface functions), לצורך גישה לאלמנטים פרטיים של המחלקה.

בפרק 21 ציינו שצריך לכלול בהגדרת המחלקה מידע רב ככל האפשר אודות העצם. בצורה זו העצמים יכילו מטען מידע המאפשר להם להיות בסיס למספר גדול של תוכניות. כך, הם יהיו **מוכללים** (**Self contained**) במידה כזו שהם יספקו את כל הדרוש, הן מבחינת הנתונים והן מבחינת הפעולות המתבצעות על נתונים אלה.

הסתרת מידע

המחלקה מורכבת מנתונים ושיטות (פונקציות). כדי להשתמש במחלקות ולנצל את יכולתן, התוכנית צריכה להכיר מה הם הנתונים שהמחלקה מסוגלת לאחסן (משתנים) ומה הן השיטות המוגדרות בה (פונקציות). לעומת זאת, אין כל צורך לדעת כיצד השיטות פועלות. לדוגמה, במחלקה מטיפוס **קובץ** (file), מספיק לדעת שקיימת בה פונקציה בשם print ושהקריאה אליה על ידי המשפט file.print תגרום להדפסת עותק מפורמט של הקובץ. בדומה, טוב לדעת על קיום הפונקציה file delete המבצעת מחיקה של קובץ. כמובן שמבחינת התוכנית אין צורך לדעת כיצד פונקציות אלו פועלות. במילים אחרות, התוכנית צריכה להתייחס אליהן כ**קופסאות שחורות**. למעשה, התוכנית מכירה את השיטות שמוגדרות במחלקה **ויודעת** לקרוא אליהן ולהעביר להן את הפרמטרים הדרושים, אך אין היא מכירה את התהליכים שמתרחשים בתוך המחלקה.

הסתרת מידע (**Information hiding**), היא תהליך בו מאפשרים לתוכנית להכיר את המידע המינימלי הדרוש לה לשם שימוש וניצול המחלקה. **החלק הפרטי** (Private) **והחלק הציבורי** (Public) של המחלקה מאפשרים ליישם את תהליך הסתרת המידע. במחלקות המוגדרות בפרק 21 הוצבה התווית "public:", כך שכל האלמנטים של המחלקה מוכרים וידועים על ידי התוכנית כולה. לכן, מכל מקום בתוכנית ניתן לגשת לאלמנטים של המחלקה תוך שימוש באופרטור הנקודה.

```
class employee {
 public:
  char name[64];
  long employee_id;
  float salary;
  void show_employee(void);
}
```

בתהליך עיצוב המחלקה ניתן לראות שחלק מהאלמנטים שלה מיועדים לשימוש פנימי, ולשאר מרכיבי התוכנית אין צורך בהם. אלמנטים אלה הם אלמנטים **פרטיים** של המחלקה וצריכים להיות מוסתרים לתוכנית. אם אין מציינים את התווית "public:" בתוך המחלקה, המשמעות היא שכל האלמנטים של המחלקה מוצבים בחלק **הפרטי** שלה, ואז אי אפשר יהיה לגשת אליהם על ידי שימוש באופרטור הנקודה. הגישה לאלמנטים בחלק הפרטי של מחלקה מותרת לפונקציות המחלקה בלבד. כאשר מגדירים את המחלקה, יש להפריד בין החלק **הפרטי** לבין החלק **הציבורי** שלה לפי הדוגמה הבאה.

```
class some_class {
  public:
    int some_variable;
    void initialize_private(int, float);
    void show_data(void);
  private:
    int key_value;
    float key_number;
}
```

משתנה ציבורי

משתנה פרטי

כפי שניתן לראות, התוויות private ו-public מאפשרות להגדיר בצורה ברורה את האלמנטים הפרטיים ואת האלמנטים הציבוריים של המחלקה. במקרה זה, תתאפשר גישה ישירה לאלמנטים **הציבוריים** של המחלקה על ידי שימוש באופרטור הנקודה, לפי הדוגמה הבאה:

```
some_class object;      // Create an object
object.some_variable = 1001;
object.initialize_private(2002, 1.2345);
object.show_data()
```

אם ננסה לגשת לאלמנטים **הפרטיים** של המחלקה key_value, או key_number תוך שימוש באופרטור הנקודה, המהדר יציג הודעה של שגיאת תחביר (Syntax error).

בדרך כלל נהוג להגן על נתוני המחלקה מפני גישה ישירה של התוכנית על ידי הצבתם בחלק הפרטי של המחלקה. על כן, התוכנית אינה יכולה לגשת ולעדכן את ערכי המשתנים האלה על ידי שימוש באופרטור הנקודה. במקום זה, שימוש או עדכון של הערכים במשתנים יתבצע על ידי קריאה לפונקציות עדכון המוגדרות במחלקה. על ידי מניעת הגישה הישירה של התוכנית אל משתני המחלקה, ניתן להבטיח שהערכים שיוצבו במשתני המחלקה תמיד יהיו תקינים. למשל, נניח כי בתוכנית מוגדר עצם מטיפוס nuclear_reactor המכיל משתנה בשם melt_down, אשר חייב לקבל ערך מתוך טווח המספרים הנע בין 1 לבין 5 בלבד. אם המשתנה melt_down הוא אלמנט ציבורי של המחלקה, התוכנית יכולה לפנות אליו באופן ישיר ולעדכן את הערך שלו:

```
nuclear_reactor.melt_down = 101
```

אילו היינו מציבים את המשתנה בחלק הפרטי של המחלקה, היינו יכולים להשתמש בשיטה המקובלת במחלקה, כמו הפונקציה assign_melt_down, שתפקידה לעדכן או לשנות את ערך המשתנה. בדוגמה ניתן לראות שהפונקציה מבצעת בדיקות תקינות של הקלט למשתנים, כך שניתן להבטיח שהנתון המאוחסן במשתנה תמיד יהיה תקין.

```
int nuke::assign_meltdown(int value)
 {
   if ((value > 0) && (value <= 5))
    {
      melt_down = value;
      return(0);  // Successful assignment
    }
   else
     return(-1);  // Invalid value
```

שיטות המחלקה המיועדות לבקרת תקניות הקלט של משתני המחלקה נקראות **פונקציות ממשק** (**Interface Functions**). פונקציות ממשק מאפשרות להגן על הנתונים של המחלקה.

אלמנטים פרטיים ואלמנטים ציבוריים

המחלקות בשפת C++ מורכבות מנתונים ומשיטות (Methods). כדי להבדיל בין האלמנטים שמותרת גישה ישירה אליהם על ידי שימוש באופרטור הנקודה, לבין האלמנטים האחרים, שפת C++ מאפשרת להגדיר במחלקה אלמנטים פרטיים ואלמנטים ציבוריים. מצד אחד, ניתן לגשת לאלמנטים הציבוריים על ידי שימוש באופרטור הנקודה. מצד שני, הגישה לאלמנטים הפרטיים מתאפשרת באמצעות פונקציות המחלקה בלבד. ככלל, מומלץ להגן על נתונים רבים ככל האפשר המוגדרים במחלקה על ידי הצבתם בחלק הפרטי של המחלקה.

אלמנטים פרטיים ואלמנטים ציבוריים

בתוכנית **INFOHIDE.CPP** מתואר השימוש באלמנטים הפרטיים ובאלמנטים הציבוריים של המחלקה. בתוכנית מוגדר עצם מטיפוס employee, שהיא המחלקה המוגדרת להלן.

```
class employee {
 public:
   int assign_values(char *, long, float);
   void show_employee(void);
   int change_salary(float);
   long get_id(void);
 private:
   char name[64];
   long employee_id;
   float salary;
}
```

כל משתני המחלקה מוגנים על ידי הגדרתם כאלמנטים פרטיים. הגישה לנתוני המחלקה מתאפשרת באמצעות פונקציות ממשק של המחלקה בלבד. להלן קוד התוכנית **INFOHIDE.CPP**.

```
#include <iostream.h>
#include <string.h>

class employee {
 public:
   int assign_values(char *, long, float);
   void show_employee(void);
   int change_salary(float);
   long get_id(void);
 private:
   char name[64];
   long employee_id;
   float salary;
};

  int employee::assign_values(char *emp_name,
   long emp_id, float emp_salary)
 {
   strcpy(name, emp_name);
   employee_id = emp_id;
   if (emp_salary < 50000.0)
    {
      salary = emp_salary;
      return(0);               // Successful
    }
   else
     return(-1);               // Invalid salary
 }

void employee::show_employee(void)
 {
   cout << "Employee: " << name << endl;
   cout << "Id: " << employee_id << endl;
   cout << "Salary: " << salary << endl;
 }

int employee::change_salary(float new_salary)
 {
   if (new_salary < 50000.0)
    {
      salary = new_salary;
```

```
      return(0);                // Successful
     }
   else
     return(-1);                // Invalid salary
 }

long employee::get_id(void)
 {
   return(employee_id);
 }

void main(void)
 {
   employee worker;

   if (worker.assign_values("Happy Jamsa", 101,10101.0)
      == 0)
   {
      cout << "Employee values assigned" << endl;
      worker.show_employee();

      if (worker.change_salary(35000.00) == 0)
       {
         cout << "New salary assigned" << endl;
         worker.show_employee();
       }
    }
   else
     cout << "Invalid salary specified" << endl;
 }
```

על אף שהתוכנית ארוכה, הפונקציות המוגדרות בה קלות להבנה, לכן כדאי לבחון את משפטי התוכנית. תפקידה של פונקציה assing_value הוא לתת ערכים התחלתיים למשתנים שנמצאים בחלק הפרטי של המחלקה. בפונקציה זו מתבצעת בדיקת תקינות קלט. הפקודה If בפונקציה בודקת אם הזנת שכר העובד חוקית. הפונקציה show_employee מציגה על המסך את הערכים המאוחסנים במשתנים הפרטיים של המחלקה. במחלקה שתי פונקציות ממשק: change_salary ו-get_id, שדרכן ניתן לגשת לאלמנטים הפרטיים של המחלקה. לאחר הידור והרצה מוצלחים, מומלץ לשנות את קוד התוכנית ולנסות לגשת למשתנים הפרטיים של המחלקה על ידי שימוש באופרטור הנקודה. את משפטי הגישה יש להציב בתוכנית הראשית main. ברגע שנבצע הידור של התוכנית, המהדר יציג על מסך הודעת שגיאה, מכיון שאין אפשרות לגשת בצורה ישירה לאלמנטים הפרטיים של המחלקה.

פונקציות ממשק

כדי למנוע שגיאות ותקלות שונות ולהכתיב גישה מסודרת לאיברי המחלקה, צריך להגביל את הגישה אל נתוני המחלקה. לשם כך מגדירים את איברי המחלקה כ**פרטיים** (Private). בדרך זו התוכנית אינה יכולה לפנות אל הנתונים שבמחלקה בעזרת אופרטור הנקודה. על המחלקה להגדיר פונקציות ממשק, שבאמצעותן ניתן יהיה להקצות ערכים ל**איבריה הפרטיים**, המוגדרים private. פונקציות הממשק יבחנו ויתנו תוקף לערכים שיוקצו לאיברים אלה.

איברי המחלקה
ואופרטור טווח ההכרה הכללי

יש לשים לב ששמות הפרמטרים של הפונקציות המוגדרות בתוכנית **INFOHIDE.CPP** מתחילים בצירוף האותיות _emp.

```
int employee::assign_values(char *emp_name, long emp_id,
  float emp_salary)
```

הצבת צירוף האותיות _emp בתחילת שמות הפרמטרים של הפונקציות מיועדת למנוע התנגשות בין שמות הפרמטרים לבין שמות המשתנים של המחלקה. כאשר סוג זה של התנגשות מתרחש, ניתן לפתור אותה על ידי השימוש באופרטור **טווח ההכרה הכללי** (**Global Resolution Operator**) - (**::**) של שפת ++C. להגדרת הפונקציה הבאה משתמשים באופרטור טווח ההכרה ומציבים אותו ואת שם המחלקה לפני שמות משתני המחלקה. בצורה זו מאוד קל להבדיל בין שמות משתני המחלקה employee לבין שמות הפרמטרים של הפונקציה.

```
int employee::assign_values(char *name,
  long employee_id,  float salary)
 {
   strcpy(employee::name, name);
   employee::employee_id = employee_id;
   if (salary < 50000.0)
    {
      employee::salary = salary;
      return(0);                // Successful
    }
   else
    return(-1);                 // Invalid salary
```

השימוש בשם המחלקה ובאופרטור טווח ההכרה בצורה המתוארת לעיל, מונע התנגשות בין שמות המשתנים ובין הפרמטרים של הפונקציה.

נצל את אופרטור טווח ההכרה לפתרון התנגשות איברי המחלקה

בעת הצהרה על פונקציות מחלקה ייתכן ששם של משתנה מקומי השייך לפונקציה (שיטה) מסוימת, זהה לשם של אחד מאיברי המחלקה. כברירת המחדל, המהדר יתייחס תמיד לשם המשתנה המקומי, אשר יגבר על השם הפרטי במחלקה. ניתן לגשת כך אל איבר המחלקה המתאים, על ידי שימוש בשם המחלקה ובאופרטור טווח ההכרה:

```
class_name::member_name = some_value;
```

הגדרת פונקציות פרטיות במחלקה

בכל הדוגמאות המוצגות בפרק הנוכחי, הוצבו המשתנים בחלק הפרטי של המחלקה. בתוכניות מורכבות יותר, ייתכן שלא יהיה רצוי שתהיה גישה ישירה מהתוכנית למספר פונקציות במחלקה. במקרה זה ניתן להציב את הגדרת הפונקציות האלו בחלק הפרטי של המחלקה, וכך למנוע גישה ישירה אליהן על ידי שימוש באופרטור הנקודה.

סיכום

בקרת הגישה של התוכנית לאלמנטים של המחלקה מקטינה את הסיכוי לתקלות הנובעות משימוש שגוי באלמנטים אלה. בקרת הגישה לאלמנטים של המחלקה מתבצעת על ידי הצבתם בחלק הפרטי שלה. ברוב המחלקות המוגדרות בשפת C++ קיים שילוב של אלמנטים פרטיים ואלמנטים ציבוריים.

לפני שנעבור לפרק הבא, נבדוק האם מובנים לנו הנושאים הבאים:

- ✓ אלמנטים במחלקה יכולים להיות בחלק הפרטי או בחלק הציבורי של המחלקה. מכל מקום בתוכנית ניתן לגשת בצורה ישירה לאלמנטים הציבוריים של המחלקה באמצעות אופרטור הנקודה. הגישה לאלמנטים הפרטיים של המחלקה נעשית באמצעות פונקציות המחלקה בלבד.
- ✓ על פי ברירת המחדל, ובהיעדר ציון מפורש, שפת C++ מניחה שכל האלמנטים במחלקה הם פרטיים.
- ✓ השמת ערכים למשתנים פרטיים במחלקה מתבצעת על ידי פונקציות ממשק.
- ✓ בפונקציות המחלקה ניתן להשתמש בשם המחלקה ובאופרטור טווח ההכרה (::) לפי הפורמט employee::name, וכך למנוע התנגשות בין שמות המשתנים ובין הפרמטרים של הפונקציה.

אחת הפעולות הנפוצות בעת יצירת עצם היא הצבה של ערכים התחלתיים במשתנים שלו. בפרק 23 נציג את **הבנאים** (**Constructors**) בשפת C++, שהם פונקציות מיוחדות שמופעלות בצורה אוטומטית בעת יצירת העצם. על ידי שימוש בפונקציית בנייה ניתן לאתחל בקלות את העצמים של המחלקה.

תרגילים

1. כיתבו מחלקה בשם person המספקת את הפונקציות הציבוריות האלו:

```
void init(char* name, int age)
void print(void);
```

הפונקציה print תדפיס את שם האדם, גילו ושנת הלידה שלו. יש להשתמש בפונקציה פרטית בשם birth_year, המחשבת את שנת הלידה על פי הגיל והשנה הנוכחית.

הערה: *השתמשו בפונקציה ()getdate לקבלת השנה הנוכחית.*

2. כיתבו מחלקה המייצגת נקודה במישור על ידי קואורדינטות x,y. משתני המחלקה x,y יהיו פרטיים. כיתבו פונקציות לאתחול הנקודה והחזרת מיקום הנקודה (הקואורדינטות).

3. כיתבו מחלקה הכוללת אוסף (set) של מספרים ממשיים. על המחלקה להגדיר את הפונקציות הציבוריות הבאות:

   - הוצאת כל המספרים מהקבוצה - void clear(void)
   - הוספת מספר לקבוצה - void add(float num)
   - בדיקה אם מספר נתון נמצא בקבוצה - int exist(float num)

   יש להוסיף מספר רק אם אינו כבר קיים באוסף. על המחלקה להשתמש במערך פרטי באורך 50 לשמירת איברי האוסף.

4. כיתבו מחלקה לביצוע הפעולות המוגדרות בתרגיל 3, אך המאפשרת מספר לא מוגבל של איברים באוסף. רצוי להשתמש ברשימה מקושרת פרטית.

הערה: *לפניך תרגיל מורכב במקצת. אם דרוש לך מידע נוסף בנושא "רשימה מקושרת", עיין בספר אחר, כמו למשל* ***"המדריך השלם לשפת C"*** *שבהוצאת הוד-עמי.*

פרק 23

פונקציות בנייה ופונקציות פירוק

אחת הפעולות הדרושות בעת יצירת העצם היא לקבוע ערכים התחלתיים למשתני העצם. בפרק 22 ראינו שפונקציות המחלקה הן הדרכים היחידות המובילות לאלמנטים הפרטיים של המחלקה. כדי לפשט את תהליך אתחול העצם, קיים בשפת C++ סוג של פונקציה מיוחדת שמופעלת כאשר העצם נוצר. פונקציה מסוג זה נקראת **פונקציית בנייה** (**Constructor function**) או **בנאי.** בשפת C++ קיים סוג נוסף של פונקציה מיוחדת שמופעלת לאחר סיום הקיום של העצם בתוכנית. פונקציה מסוג זה נקראת **פונקציית פירוק** (**Destructor function**) או **מפרק**.

בהמשך הפרק נדון בנושאים הבאים:

- פונקציות בנייה הן שיטות מחלקה (Methods), המקלות על אתחול משתני המחלקה. השם של פונקציות הבנייה זהה לשל המחלקה.
- פונקציות בנייה אינן מחזירות ערך, או טיפוס.
- בכל יצירה של משתנה מחלקה, מופעלת אוטומטית פונקציית הבנייה של המחלקה, אם קיימת כזו.
- עצמים שונים עשויים להקצות זיכרון לאחסנת נתונים. כאשר מוחקים עצם, C++ קוראת לפונקציית הפירוק של המחלקה, אשר "מנקה" את המקום בזיכרון שתפסו עד עתה הנתונים של העצם.
- השם של פונקציית פירוק זהה לשם המחלקה לה היא שייכת, ולפניו מופיע הסימן **~** (tilde).
- פונקציות פירוק אינן מחזירות ערך.

ייתכן שהמונחים **בנייה** ו**פירוק** יתפרשו באופן מוטעה לאוזן הלא מקצועית. יש לפרשם במובן של כלי עזר בתכנות. הבנאי עוזר להרכבה של העצם, ולעומת זאת המפרק, על פי הגדרתו, פועל בשלבי סיום פעולתו של העצם. במקרים רבים, תפקיד המפרק הוא לשחרר את הזיכרון שנתפס על ידי העצם בזמן שפעל.

יצירת פונקציית בנייה

פונקציית הבנייה (Constructor function) היא פונקציית מחלקה הנושאת את שם המחלקה. לדוגמה, אם שם המחלקה הוא employee, בנאי המחלקה נקרא employee. אנלוגית, במחלקה dogs, שם הבנאי יהיה גם כן dogs. כאשר בתוכנית מסוימת מוגדרת פונקציית בנייה, שפת C++ מריצה את הפונקציה הזו בצורה אוטומטית בעת יצירת העצם. בתוכנית **CONSTRUC.CPP** המוצגת כאן, מוגדרת מחלקה בשם employee. בנוסף, בתוכנית מוגדר בנאי בשם employee שתפקידו לקבוע ערכים התחלתיים למשתני העצם. פונקציית בנייה אינה מחזירה ערך, ולמרות זאת אין להגדיר אותה באמצעות void. הפונקציה מוגדרת ללא ציון טיפוס של ערך מוחזר.

```
class employee {
 public:
   employee(char *, long, float);
      // Constructor function (no return type)
   void show_employee(void);
   int change_salary(float);
   long get_id(void);
 private:
   char name[64];
   long employee_id;
   float salary;
};
```

הכללים להגדרה של פונקציית הבנייה בתוכנית זהים לכללי ההגדרה של פונקציית מחלקה כלשהי:

```
employee::employee(char *name, long employee_id,
  float salary)
 {
   strcpy(employee::name, name);
   employee::employee_id = employee_id;
   if (salary < 50000.0)
     employee::salary = salary;
   else
     // Invalid salary specified
     employee::salary = 0.0;
 }
```

ניתן לראות שפונקציית הבנייה אינה מחזירה ערך לפונקציה הקוראת. במקרה זה, פונקציית הבנייה משתמשת באופרטור טווח ההכרה הכללי ובשם המחלקה לפני כל אלמנט, כפי שלמדנו בפרק הקודם. להלן יישום מלא של התוכנית **CONSTRUC.CPP**.

```
#include <iostream.h>
#include <string.h>

class employee {
 public:
   employee(char *, long, float);
   void show_employee(void);
   int change_salary(float);
   long get_id(void);
 private:
   char name[64];
   long employee_id;
   float salary;
};

employee::employee(char *name, long employee_id,
  float salary)
 {
   strcpy(employee::name, name);
   employee::employee_id = employee_id;
   if (salary < 50000.0)
     employee::salary = salary;
   else
     // Invalid salary specified
     employee::salary = 0.0;
 }

void employee::show_employee(void)
 {
   cout << "Employee: " << name << endl;
   cout << "Id: " << employee_id << endl;
   cout << "Salary: " << salary << endl;
 }

void main(void)
 {
   employee worker("Happy Jamsa", 101, 10101.0);

   worker.show_employee();
 }
```

יש לשים לב שהצהרה על העצם worker כוללת את הערכים ההתחלתיים של נתוני העצם המסודרים בתוך סוגריים, בדומה לדרך העברת פרמטרים לפונקציה רגילה. הנתונים שבתוך הסוגריים מועברים כפרמטרים לפונקציות הבנייה של מחלקת העצם.

```
employee worker("Happy Jamsa", 101, 10101.0);
```

בתוכנית ניתן ליצור מספר עצמים מטיפוס employee, אשר לכל אחד מהם ניתן להעביר את הפרמטרים הדרושים לבנאי כדי לבצע את אתחול העצם. להלן דוגמאות של משפטים ליצירת עצמים מטיפוס employee:

```
employee worker("Happy Jamsa", 101, 10101.0);
employee secretary("John Doe", 57, 20000.0);
employee manager("Jane Doe", 1022, 30000.0:
```

פונקציית בנייה, מה היא ?

פונקציית בנייה (Construction function) היא פונקציה מיוחדת שמופעלת בצורה אוטומטית בכל פעם שנוצר עצם מטיפוס המחלקה. לרוב משתמשים בפונקציית הבנייה לאתחול משתני העצם. שם פונקציית הבנייה זהה לשם המחלקה לה היא שייכת. למשל, במחלקה ששמה file, שם פונקציית הבנייה יהיה גם כן file. אופן הגדרת הבנאי דומה לאופן הגדרת פונקציית מחלקה רגילה, למעט העובדה שבפונקציית הבנייה לא מציינים את טיפוס הערך המוחזר. בעת יצירת עצם במחלקה ניתן להעביר פרמטרים לפונקציית הבנייה באמצעות משפט יצירת העצם, לפי התבנית הבאה:

```
class_name object(value1, value2, value3)
```

ערכי מחדל של פרמטרים בפונקציית בנייה

בשפת ++C ניתן לקבוע ערכי ברירת מחדל לפרמטרים בפונקציה. ערכים אלה יופעלו כאשר במשפט הקריאה לפונקציה לא יועברו ערכים לפרמטרים מסוימים. תכונה זו הוצגה בפרקים קודמים בספר, והיא ניתנת ליישום גם בפונקציות בנייה. לדוגמה, בפונקציית הבנייה employee המוצגת בהמשך, נקבע לפרמטר salary ערך ברירת מחדל. אם לא יועבר ערך לפרמטר זה, הפרמטר יקבל מפונקציית הבנייה את הערך 10000.0. לעומת זאת, את שם העובד ומספר זיהוי חייבים להעביר לפונקציה.

```
employee::employee(char *name, long employee_id,
  float salary = 10000.00)
 {
   strcpy(employee::name, name);
   employee::employee_id = employee_id;
   if (salary < 50000.0)
     employee::salary = salary;
   else
     // Invalid salary specified
     employee::salary = 0.0;
 }
```

העמסה של פונקציות בנייה

בשפת ++C ניתן לבצע העמסה של פונקציות על ידי הגדרת פונקציות בעלות שם זהה ורשימת פרמטרים שונה. תכונה זו ניתנת ליישום גם בפונקציות בנייה. בתוכנית **CONSOVER.CPP** מתוארת העמסה של פונקציית הבנייה employee. בהגדרה הראשונה של הפונקציה מוגדרים בה כפרמטרים שם, מספר זיהוי ושכרו של העובד. פרמטרים אלה חייבים לעבור במשפט היצירה של העצם. בהגדרה השנייה של הפונקציה המשתמש נדרש להזין את שכר העובד, כאשר נתון זה לא מועבר לעצם בעת יצירתו.

```
employee::employee(char *name, long employee_id)
 {
   strcpy(employee::name, name);
   employee::employee_id = employee_id;
   do {
     cout << "Enter a salary for " << name <<
                                  " less than $50,000: ";
     cin >> employee::salary;
   } while (salary >= 50000.0);
 }
```

בתוך הגדרת המחלקה מוגדרים גם שני הבנאים על ידי תבניות הפונקציה שלהם.

```
class employee {
 public:
  employee(char *, long, float);
      employee(char *, long);
  void show_employee(void);
  int change_salary(float);
  long get_id(void);
 private:
   char name[64];
   long employee_id;
   float salary;
}
```

תבניות של פונקציות מועמסות

להלן יישום מלא של התוכנית **CONSOVER.CPP**:

```
#include <iostream.h>
#include <string.h>

class employee {
 public:
   employee(char *, long, float);
   employee(char *, long);
   void show_employee(void);
   int change_salary(float);
   long get_id(void);
```

```
 private:
   char name[64];
   long employee_id;
   float salary;
};

employee::employee(char *name, long employee_id,
  float salary)
 {
   strcpy(employee::name, name);
   employee::employee_id = employee_id;
   if (salary < 50000.0)
     employee::salary = salary;
   else
     // Invalid salary specified
     employee::salary = 0.0;
 }

employee::employee(char *name, long employee_id)
 {
   strcpy(employee::name, name);
   employee::employee_id = employee_id;

   do {
     cout << "Enter a salary for " << name <<
                                 " less than $50,000: ";
     cin >> employee::salary;
   } while (salary >= 50000.0);
 }

void employee::show_employee(void)
 {
   cout << "Employee: " << name << endl;
   cout << "Id: " << employee_id << endl;
   cout << "Salary: " << salary << endl;
 }

void main(void)
 {
   employee worker("Happy Jamsa", 101, 10101.0);
   employee manager("Jane Doe", 102);

   worker.show_employee();
   manager.show_employee();
 }
```

הרצת התוכנית תתחיל בבקשה להזנת שכרה של העובדת Jane Doe. לאחר הזנת הערך הדרוש, התוכנית תמשיך ותציג על המסך את נתוני שני העובדים.

יצירת פונקציית פירוק

פונקציית פירוק (**Destructor function**) מופעלת מייד לאחר שעצם בתוכנית מסיים את תפקידו - **נהרס** (Destroyed). עצם מסיים את תפקידו בסוף התוכנית, או בסוף טווח הכרתו, על פי חוקי הגדרת משתנה. בפרקים הבאים נציג כיצד ליצור רשימת עצמים המופעלים בעת שהתוכנית רצה. לצורך משימה זו יש להקצות שטח בזיכרון המחשב לאחזקת העצמים (טרם הסברנו איך מבצעים הקצאה של זיכרון). במשך ביצוע התוכנית יוצרים ומפרקים עצמים, ולכן משימה זו משתלבת כראוי עם השימוש בפונקציות פירוק.

בתוכניות שהוצגו עד כאן, התוכנית יצרה את העצמים מייד לאחר שהופעלה, בכך היא הצהירה עליהם. בסיום התוכנית העצמים נהרסים. כאשר מגדירים פונקציית פירוק, היא מופעלת אוטומטית על ידי ++C עבור כל עצם בשעה שהתוכנית מסיימת את פעולתה (כאשר העצמים נהרסים). בדומה לפונקציית בנייה, שם פונקציית הפירוק זהה לשם המחלקה שהיא שייכת אליה. לפני שם המפרק יש להציב את הסימן "~", לפי התבנית הבאה:

```
~class_name(void)                    ~ indicates destructor
  {
    // Function statements
  }
```

פונקציית פירוק אינה מקבלת פרמטרים כלל.

בתוכנית **DESTRUCT.CPP** משולבת הגדרת פונקציית הפירוק הבאה. פונקציה זו שייכת למחלקה employee.

```
void employee::~employee(void)
 {
   cout << "Destroying the object for " << name << endl;
 }
```

במקרה זה, הפונקציה מציגה על המסך הודעה על פירוק העצם. בסיום התוכנית, שפת ++C קוראת בצורה אוטומטית לפונקציית הפירוק עבור כל עצם. להלן יישום מלא של התוכנית **DESTURCT.CPP**.

```
#include <iostream.h>
#include <string.h>

class employee {
 public:
   employee(char *, long, float);
   ~employee(void);
   void show_employee(void);
   int change_salary(float);
   long get_id(void);
```

```
 private:
   char name[64];
   long employee_id;
   float salary;
};

employee::employee(char *name, long employee_id,
  float salary)
 {
   strcpy(employee::name, name);
   employee::employee_id = employee_id;
   if (salary < 50000.0)
     employee::salary = salary;
   else
     // Invalid salary specified
     employee::salary = 0.0;
 }

employee::~employee(void)
 {
   cout << "Destroying the object for " << name << endl;
 }

void employee::show_employee(void)
 {
   cout << "Employee: " << name << endl;
   cout << "Id: " << employee_id << endl;
   cout << "Salary: " << salary << endl;
 }

void main(void)
 {
   employee worker("Happy Jamsa", 101, 10101.0);

   worker.show_employee();
 }
```

לאחר הידור והרצת התוכנית נראה על המסך את השורות הבאות:

```
C:\> DESTRUCT  <Enter>
Employee: Happy Jamsa
Id: 101
Salary: 10101
Destroying the object for Happy Jamsa
```

יש לשים לב שהתוכנית מפעילה את פונקציית הפירוק לאחר סיום התוכנית, מבלי לקרוא לפונקציה. לעת עתה אין משמעות רבה לפונקציות פירוק. בפרקים הבאים נלמד כיצד להקצות זיכרון במחשב לשם ביצוע משימות בעצם, ונגלה את היכולת ואת נוחות השימוש בפונקציות פירוק לשחרור זיכרון המחשב כאשר הורסים, או מפרקים את העצם.

פונקציית פירוק מה היא?

פונקציית פירוק (Destructor function) היא פונקציה מיוחדת שמופעלת אוטומטית בכל פעם שמפרקים את העצם. שם פונקציית הפירוק זהה לשם המחלקה לה היא שייכת. לפני שם פונקציית הפירוק יש להציב את הסימן "~", לדוגמה: employee~. אופן הגדרת פונקציית הפירוק דומה לאופן הגדרת פונקציית מחלקה רגילה.

סיכום

בנאים ומפרקים הם פונקציות מיוחדות שמתבצעות בצורה אוטומטית בעת יצירת העצם, או כאשר מפרקים את העצם. רוב התוכניות נעזרות בפונקציות בנייה לאתחול משתני העצם. לרוב התוכניות הפשוטות, וביניהן אלו שהצגנו בפרק הנוכחי, אינן זקוקות לפונקציות פירוק.

לפני שנעבור לפרק הבא, נבחן אם מובנים לנו הנושאים הבאים:

- ✓ פונקציית הבנייה היא פונקציה מיוחדת הנקראת באופן אוטומטי כל פעם שנוצר עצם. היא מקבלת את שם המחלקה אליה היא מתייחסת.
- ✓ פונקציות בנייה אינן מחזירות ערך, ולמרות זאת אין להגדיר אותן מטיפוס void. פונקציות אלו מוגדרת ללא ציון טיפוס ערך מוחזר.
- ✓ בעת יצירת עצם, התוכנית יכולה להעביר פרמטרים לפונקציית הבנייה.
- ✓ שפת C++ מאפשרת להעמיס פונקציות בנייה ולקבוע ערך ברירת מחדל עבור הפרמטרים שלה. פונקציית הפירוק היא פונקציה מיוחדת הנקראת באופן אוטומטי כל פעם שעצם מתפרק (או נהרס). פונקציות הפירוק מקבלות את אותו השם של המחלקה אליהן הן מתייחסות. לפני השם הפונקציה יש להציב את הסימן "~".

העמסת אופרטורים מאפשרת לדוגמה, להגדיר מחדש את פעולת הסימן (+), כדי שנוכל באמצעותו לחבר מחרוזת מסוימת למחרוזת נתונה. בתחילת הספר למדנו שטיפוס נתון (כגון int, float, char וכו') מגדיר את תחום הערכים שהמשתנה המוגדר יכול לאחסן בתוכו. בנוסף, הטיפוס הנתון קובע את הפעולות שניתן לבצע עם אותם המשתנים. הגדרת המחלקה במהותה היא הגדרת טיפוס כיון ששפת C++ מאפשרת לקבוע מחדש את פעולת האופרטורים עבור נתוני המחלקה.

תרגילים

1. כיתבו תוכנית המיישמת שימוש במפתח סודי, המופעל בעזרת מחלקה Key. ממשק המחלקה הוא:

```
class Key {
 public:
   Key(void);
   Key(long);
   void set(void);
   void certificate(void);
   ....
   };
```

- ❖ Key(void) - אתחול ערך המפתח לערך המציין שלמפתח עוד לא נקבע ערך אמיתי.
- ❖ Key(long) - אתחול ערך המפתח בהתאם לארגומנט.
- ❖ set - קביעת ערך חדש למפתח, אחרי שמתבצע אימות, בו מתברר שהמשתמש אכן יודע מה ערכו הנוכחי של המפתח, או לחילופין, המפתח עדיין לא אותחל.
- ❖ certificate - ייצור תעודה המכריזה כי מחזיק תעודה זו הוא בעליו החוקיים של מפתח בעל ערך XXXX (הערך XXXX נקבע על פי ערכו הנוכחי של המפתח), אחרי שמתבצע אימות, בו מתברר שהמשתמש אכן יודע מה ערכו הנוכחי של המפתח.

2. כיתבו מחלקה Point המייצגת נקודה במישור. הבנאי (constructor) מאתחל את הנקודה (ברירת המחדל היא 0,0 -) ומודיע על היווצרותה. המפרק (destructor) מודיע על השמדתה של הנקודה. פונקציית המחלקה print מדפיסה את ערך הנקודה. השתמשו בפונקציה main הבאה, כדי לבחון את תפקוד המחלקה שכתבתם:

```
void main(void)
  {
    Point p1(5.5,7.3), p2(5.2), p3(0,4.4), p4;

    p1.print();
    p2.print();
    p3.print();
    p4.print();

    cout << endl;
  }
```

3. כיתבו מחלקה tree המנהלת עץ בינארי של ערכים שלמים (int). התכונה המאפיינת את העץ: אם מסתכלים על צומת נתון, אזי כל הצמתים בתת-העץ השמאלי שלו הם בעלי ערכים קטנים משלו; וכל הצמתים בתת-העץ הימני שלו, הם בעלי ערכים גדולים משלו. המחלקה תתמוך בפעולות אלו: הכנסת ערך לעץ, חיפוש ערך בעץ (האם נמצא - כן/לא), מציאת המקסימום בעץ (העלה הימני ביותר), מציאת המינימום בעץ (העלה השמאלי ביותר), בירור מספר הצמתים בעץ, בירור גובה העץ, וגם - בנאי ומפרק מתאימים.

הערה: *תרגיל זה קשה במעט, ולמי שאינו מכיר את נושא העץ הבינארי ורקורסיה מומלץ לעיין בספר "**המדריך השלם לשפת C**" בהוצאת הוד-עמי.*

4. כיתבו מחלקה chain המנהלת שרשרת של אובייקטים באורך N הנקבע עם יצירתה. לאובייקט J בשרשרת יהיה מספר סידורי J והוא יצביע אל האובייקט J-1. האובייקט עם מספר סידורי 0 יצביע ל-Null. המחלקה גם תתמוך בפונקציית מחלקה print, שבה האובייקט J מדפיס הודעה נתונה (כארגומנט) ומפעיל באופן רקורסיבי את ההדפסה של האובייקט J-1 (עד לאובייקט 0). כיתבו גם בנאי ומפרק מתאימים עבור המחלקה chain.

פרק 24

העמסת אופרטורים

בפרקים הקודמים למדנו שטיפוס המשתנה קובע את הערכים שניתן לאחסן בו, ואת הפעולות החוקיות שניתן לבצע בו. לדוגמה, משתנה מטיפוס int מאפשר ביצוע פעולות אריתמטיות כגון: חיבור, חיסור, כפל וחילוק. לעומת זאת, אין כל משמעות לדוגמה, לאופרטור החיבור (+) כאשר האופרנדים הם מחרוזות. הגדרת המחלקה במהותה היא הגדרת טיפוס, מכיון ששפת C++ מאפשרת תכנון מחודש של פעולת האופרטורים בהתאם לנתוני המחלקה. **העמסת אופרטורים** (**Operator overloading**) היא הדרך בה משנים את משמעות האופרטור עבור מחלקה מסוימת. בפרק זה נגדיר מחלקה string ונעמיס את האופרטורים פלוס ומינוס. עבור עצמים מטיפוס string, העמסת האופרטור תגרום להוספת התווים של המחרוזת החדשה אל סוף המחרוזת הקיימת. בדרך דומה, האופרטור מינוס יגרום להורדת התווים שנמצאים במחרוזת נתונה מתוך המחרוזת הקיימת.

בהמשך הפרק נדון בנושאים הבאים:

- העמסת אופרטורים גורמת לכך שהתוכנית תהיה קריאה יותר. על כן, לא רצוי להעמיס אופרטור, אם הדבר יגרום לכך שהתוכנית תהיה פחות ברורה ומובנת.
- המילה השמורה **operator** משמשת להעמסת אופרטורים.
- כאשר מעמיסים אופרטור, יש לציין פונקציה שאליה תפנה C++ בכל פעם שתשתמש באופרטור המועמס.
- כאשר מעמיסים אופרטור עבור מחלקה מסוימת, הוא יפעל כאופרטור מועמס **עבור מחלקה זו בלבד**. שאר חלקי התוכנית ימשיכו להשתמש באופרטור על פי ההגדרה המקורית שלו.
- ניתן להעמיס כל אופרטור, מלבד ארבעה, המפורטים בטבלה 24.1.

העמסת אופרטורים מפשטת את הגדרת הפעולות הנפוצות במחלקה ומקלה על הבנת התוכנית. לכן, מומלץ להשקיע זמן בלימוד מפורט של התוכניות המוצגות בפרק זה.

העמסת האופרטורים פלוס ומינוס

העמסת אופרטור והתאמתו לנתוני המחלקה אינו משפיע על פעולתם של שאר טיפוסי הנתונים המוגדרים בשפת ++C. למשל, אם נעמיס את האופרטור **פלוס** עבור נתוני המחלקה string ניתן להמשיך ולהשתמש בו כרגיל לחיבור של שני מספרים. מהדר שפת ++C קובע איזו פעולה להפעיל לפי טיפוס הנתון. בהמשך נגדיר את המחלקה string. מחלקה זו כוללת משתנה אחד מטיפוס מחרוזת וסדרה של פונקציות, שבשלב זה אין הגדרה של אופרטורים כלשהם.

התוכנית **STRCLASS.CPP** מתבססת על המחלקה string לצורך יצירת שני עצמים מטיפוס מחלקה זו.

```
#include <iostream.h>
#include <string.h>

class string {
 public:
   string(char *);  // Constructor
   void str_append(char *);
   void chr_minus(char);
   void show_string(void);
 private:
   char data[256];
};

string::string(char *str)
 {
   strcpy(data, str);
 }

void string::str_append(char *str)
 {
   strcat(data, str);
 }

void string::chr_minus(char letter)
 {
   char temp[256];
   int i, j;

   for (i = 0, j = 0; data[i]; i++)
     // Is the letter to remove?
     if (data[i] != letter)
       // If not assign it to temp
       temp[j++] = data[i];
   temp[j] = NULL;  // End of temp
```

```
   // Copy temp's contents back to data
   strcpy(data, temp);
 }

void string::show_string(void)
 {
   cout << data << endl;
 }

void main(void)
 {
   string title("Rescued By C++");
   string lesson("Understanding Operator Overloading");

   title.show_string();
   title.str_append(" rescued me!");
   title.show_string();

   lesson.show_string();
   lesson.chr_minus('n');
   lesson.show_string();
 }
```

לצורך העמסת אופרטור משתמשים במילה השמורה operator כדי לציין את האופרטור והשיטות (פונקציות) הקשורות אליו. בהגדרת המחלקה הבאה נשתמש במילה השמורה operator לשיוך האופרטורים פלוס ומינוס לפונקציות str_appends ו-_chr בהתאמה.

```
class string {
 public:
   string(char *);  // Constructor
   void operator +(char *);
   void operator -(char);
   void show_string(void);
 private:
   char data[256];
};
```

הגדרת אופרטורים של המחלקה

בדרך זו, העמסת האופרטורים **פלוס** ו**מינוס** נעשית בתוך המחלקה. בהמשך, חייבים להגדיר את הפונקציות המיישמות את הפעולות החדשות של האופרטורים.

להלן הגדרת הפונקציה של אופרטור פלוס:

```
void string::operator +(char *str)
 {
   strcat(data, str);
 }
```

בתוכנית **OPOVERLD.CPP** מתואר השימוש בפונקציות המגדירות מחדש את פעולת האופרטורים **פלוס ומינוס**:

```
#include <iostream.h>
#include <string.h>

class string {
 public:
   string(char *);  // Constructor
   void operator +(char *);
   void operator -(char);
   void show_string(void);
 private:
   char data[256];
};

string::string(char *str)
 {
   strcpy(data, str);
 }

void string::operator +(char *str)
 {
   strcat(data, str);
 }

void string::operator -(char letter)
 {
   char temp[256];
   int i, j;

   for (i = 0, j = 0; data[i]; i++)
     if (data[i] != letter)
       temp[j++] = data[i];

   temp[j] = NULL;
   strcpy(data, temp);
 }
void string::show_string(void)
 {
   cout << data << endl;
 }
```

```
void main(void)
 {
   string title("Rescued By C++");
   string lesson("Understanding Operator Overloading");

   title.show_string();
   title + " rescued me!";
   title.show_string();

   lesson.show_string();
   lesson - 'n';
   lesson.show_string();
 }
```

נחזור ונראה את המשפטים בתוכנית שבהם מופעלים האופרטורים המועמסים:

```
// Append the text "rescued me!"
title + " rescued me!";
lesson - 'n'; // Remove the letter 'n'
```

על אף שתחביר המשפטים חוקי ותקין, הוא נראה קצת משונה. בדרך כלל רגילים להפעיל את האופרטור בביטוי אשר מחזיר ערך כלשהו, כמו במשפט זה, לדוגמה:

```
some_str = title + "text";
```

שפת C++ מעניקה חופש פעולה רחב מספיק, המאפשר למתכנת לקבוע כיצד האופרטור יתנהג בתוכנית.

במקרה זה, תחביר האופרטור **פלוס** (+) שונה ואף מוזר, למרות שהוא חוקי מבחינת כללי הכתיבה של C++. בדרך כלל נוהגים להשתמש באופרטור **פלוס** בביטוי אשר מחזיר ערך, כמו לדוגמה במשפט: some_str = title + "text";. כאשר מתכנתים ב- C++, ניתן להגדיר ולקבוע את התנהגות האופרטורים במידה גדולה של חופשיות. עם זאת, הבה נזכור שמטרת העמסת אופרטורים היא הפיכת התוכנית לברורה וקלה יותר להבנה. בהתאם לכך, התוכנית **STR_OVER.CPP** המוצגת להלן שונה במקצת מהתוכנית הקודמת. היא מאפשרת פעולות על משתני מחרוזת מטיפוס string באמצעות התחביר האופייני לפעולות אופרטור ההשמה (ראה להלן).

```
#include <iostream.h>
#include <string.h>

class string {
  public:
       string (char  *);  // Constructor
              char * operator +(char *);
              char * operator -(char);
              void show_string(void);
```

```
   private:
      char  data[256];
  };

string::string(char *str)
  {
      strcpy(data, str);
  }

char * string::operator +(char *str)
  {
      return(strcat(data, str));
  }

char * string::operator -(char letter)
  {
      char temp[256];
      int i, j;

      for (i = 0, j = 0; data[i]; i++)
        if (data[i]  != letter)
          temp[j++] = data [i];
      temp[j] = NULL;

      return(strcpy(data, temp));
  }

void string::show_string(void)
  {
      cout << data << endl;
  }

void main(void)
  {
     string title("Rescued By C++");
     string lesson("Understanding Operator Overloading");
     title.show_string();
     title = title + " rescued me!";
     title.show_string();

     lesson.show_string();
     lesson = lesson - 'n';
     lesson.show_string();
  }
```

על ידי שינוי האופרטורים **פלוס** ו**מינוס** (+, **-**) המועמסים, כדי שיחזירו מצביעים למחרוזות תווים, התוכנית יכולה להשתמש בהם על פי התחביר של משפט השמה:

```
title = title + " rescued me!";
lesson = lesson - 'n';
```

דוגמה נוספת

השוואה בין ערכים מטיפוס המחלקה היא פעולה מקובלת כאשר נעזרים במחלקה ככלי ליצירת טיפוסי נתון חדשים. לצורך זה ניתן להעמיס את האופרטור **שווה** (==), **אינו שווה** (=!), או כל אופרטור השוואה (יחס) אחר. בתוכנית **COMP_STR.CPP** מתוספת הגדרה של אופרטור במחלקה string. האופרטור מאפשר לבדוק אם שני עצמים מטיפוס string הינם שווים. על ידי העמסת האופרטור ניתן לבדוק אם שני העצמים מכילים את אותה מחרוזת:

```
if (some_string == another_string)
```

בעמודים הבאים כתובה דוגמה ליישום מלא של התוכנית **COMP_STR.CPP**:

```
#include <iostream.h>
#include <string.h>

class string {
 public:
   string(char *);  // Constructor
   void operator +(char *);
   void operator -(char);
   int operator ==(string);
   void show_string(void);
 private:
   char data[256];
};

string::string(char *str)
 {
   strcpy(data, str);
 }
char * string::operator +(char *str)
 {
   strcat(strcat(data, str));
 }

char * string::operator -(char letter)
 {
   char temp[256];
   int i, j;
```

```
   for (i = 0, j = 0; data[i]; i++)
     if (data[i] != letter)
       temp[j++] = data[i];

   temp[j] = NULL;

   strcpy(strcpy(data, temp));
 }

int string::operator ==(string str)
 {
   int i;

   for (i = 0; data[i] == str.data[i]; i++)
     if ((data[i] == NULL) && (str.data[i] == NULL))
       return(1);  // Equal

   return(0);      // Not equal
 }

void string::show_string(void)
 {
   cout << data << endl;
 }

void main(void)
 {
   string title("Rescued By C++");
   string lesson("Understanding Operator Overloading");
   string str("Rescued By C++");

   if (title == lesson)
     cout << "title and lesson are equal" << endl;

   if (str == lesson)
     cout << "str and lesson are equal" << endl;

   if (title == str)
     cout << "title and str are equal" << endl;
 }
```

דרך העמסת אופרטורים המתוארת בתוכנית הקודמת מקלה על הבנת התוכנית ומפשטת אותה.

אופרטורים שאינם ניתנים להעמסה

בתוכנית בשפת ++C ניתן להעמיס את רוב האופרטורים. עם זאת, בטבלה 24.1 מוצגת רשימת אופרטורים שאינם ניתנים להעמסה.

טבלה 24.1: אופרטורים שאינם ניתנים להעמסה בתוכנית ++C.

אופרטור	תפקיד	דוגמה
.	אלמנט (Member) במחלקה	object.member
*.	מצביע לאלמנט במחלקה	object.*member
::	טווח ההכרה הכללי	classname::member
:?	משפט (ביטוי) תנאי מקוצר	c = (a > b) ? a : b;

העמסת אופרטורים אונאריים

בהעמסת אופרטורים אונאריים, פונקציית האופרטור השייכת למחלקה לא צריכה לקבל שום ארגומנט כפרמטר מכיון שהיא תפעל רק על הנתונים של האובייקט שהפעיל אותה.

בדוגמה להלן יש לשים לב להבדל בין pre increment ל- post increment.

דוגמה : **oprunar.cpp**

הערה *: התוכנית נבדקה והורצה ב- Visual C++ 6.0.*

```
#include <iostream.h>

class circle
   {
     private:
        int radius;
     public:
        circle(int r) {radius = r;}
        circle() {radius = 0;}
        circle &operator++();
        circle operator++(int);
        void show() { cout << "radius = " << radius << endl; }
   };

circle &circle::operator++() // pre increment
   {
     radius++;
     return *this;
   }
```

```
circle circle::operator++(int)  // post increment
   {  // the (int) is just for the different between pre and post
      circle tmp = *this;
      radius++;
      return tmp;          // before the post action
   }

void main(void)
   {
      circle c1(3), c2;

      cout << "c1 first: ";
      c1.show();
      cout << "c2 = c1++: ";
      c2 = c1++;
      c2.show();
      cout << "c2 = ++c1: ";
      c2 = ++c1;
      c2.show();
   }
```

העמסת אופרטורים באמצעות פונקציות מטיפוס friend

פונקציה מסוג friend הינה פונקציה בעלת היתר גישה לכל משתני המחלקה. מומלץ לתת עדיפות לפונקציות המחלקה כדי לקיים את עקרון הריכוזיות, אולם קיים מקרה אחד אחרון בו נתקשה לטפל במסגרת פונקציות המחלקה והוא:

```
7 + object ;
```

מצד שמאל של האופרטור נמצא משתנה מטיפוס מערכת – int.

לא ניתן להגדיר פונקציית מחלקה שתגדיר מקרה זה באופן הבא:

```
7.operator + object
```

הפתרון לבעיה הוא פונקציית חבר שתכיר את משתני המלבן:

```
class object
{
    ...
    friend object & operator+(int number, const object &o);
};
```

נניח שעבור המקרה ההפוך (object + 7) העמסנו פונקציית מחלקה. אזי נממש את פונקציית החבר באופן הבא:

```
friend object & operator+(int number, const object &o)
{
```

כאן נפעיל את פונקציית המחלקה הממשת את -

```
return o + number ; // object + 7
```

דוגמה : **oprfrnd.cpp**

הערה *: התוכנית נבדקה והורצה ב- Visual C++ 6.0.*

```
#include <iostream.h>

class object
{
  private:
    int number;
  public:
    object(){};
    object(int n){number = n;}
    object operator+(int n);       // o1 = o2 + 5
    object operator+(object &o);  // o1 = o2 + o3
    friend object operator+(int n, object &o); // o1 = 7 + o2
    void show(){ cout << "number = " << number << endl; }
  };
object object::operator+(int n)
{
    object tmp;
      tmp.number = number + n;
    return(tmp);
}
object object::operator+(object &o)
{
   object tmp;
   tmp = o + number ;  // operator+(int n) כאן מופעל האופרטור
   return(tmp);
}
object operator+(int n, object &o)
{
   object tmp;
   tmp = o + n;      // operator+(int n) כאן מופעל האופרטור
   return tmp;
}

void main(void)
{
   object o1(5), o2, o3;
   o2 = o1 + 7;
   o2.show();
   o2 = 7 + o1;
   o2.show();
   (o1 + o2).show();
}
```

העמסת האופרטור []

העמסת אופרטור ה- [ ] נוחה לשימוש ומוסיפה לקריאות, לכן במקום להשתמש בפונקציות להשמת ערך/שליפת ערך ממערך המיוצג על ידי מחלקה, נשתמש בהעמסה.

תחביר העמסת האופרטור [ ]:

```
טיפוס_המערך & שם_המחלקה::operator[](int index)
       {
              גוף ההגדרה;
       }
```

דוגמה : **oprbrkts.cpp**

הערה *: התוכנית נבדקה והורצה ב- Visual C++ 6.0.*

```
#include <iostream.h>
#include <stdlib.h>

class arrayC
   {
      float *fp;
      int size;
      public:
         arrayC(int number_of_elements);
         float &operator[](int index) const;
         ~arrayC();
   };

arrayC::arrayC(int number_of_elements)
   {
      if ((fp = new float[number_of_elements])!=NULL)
         size = number_of_elements;
      else
         size = 0;
   }

float &arrayC::operator[](int index) const
   {
      if(index >= 0 && index< size)
         return(fp[index]);
      else
         {
            cout<<"index is out of range\n";
            exit(1);
         }
   }
```

```
arrayC::~arrayC()
   {
      if ( fp ) delete fp;
   }

void main(void)
   {
      arrayC  fa(4);

      fa[1] = 12.86f;
      cout << "fa[1]=" << fa[1] << endl;
      fa[3] = fa[1] + 3.21f;
      cout << " fa[3] = fa[1] + 3.21f ="  << fa[3] << endl;
   }
```

סיכום

העמסת אופרטור היא שיטה להקנות משמעות חדשה לאופרטור שמשתמשים בו במחלקה מסוימת. העמסת אופרטורים עשויה להקל על הבנת התוכניות, מכיון שאפשר לבטא פעולות הקשורות לעצמי המחלקה בצורה יותר מפורשת וברורה.

לפני שנעבור לפרק הבא, נבחן אם מובנים לנו הנושאים הבאים:

- ✓ כדי להעמיס אופרטור, צריך להגדיר מחלקה שאליה הוא יהיה קשור.
- ✓ כאשר מעמיסים אופרטור, ההעמסה פועלת רק בעצמים מטיפוס המחלקה שבה הוגדרה העמסת האופרטור. כאשר האופרטור מופעל על משתנים שאינם שייכים למחלקה, ישוב השימוש שלו למקור.
- ✓ יש להשתמש במילה שמורה operator להגדרת פונקציית המחלקה אשר האופרטור קשור אליה.
- ✓ שפת ++C אינה מאפשרת העמסת אופרטור נקודה (**.**), מצביע לאלמנט המחלקה (**.***), אופרטור טווח ההכרה (**::**) ואופרטורים של משפט תנאי מקוצר (**?:**).

בפרק 25 נלמד כיצד לשתף מידע בין עצמים על ידי פונקציות סטטיות (Static), וכיצד להשתמש בפונקציות המחלקה (Methods), כאשר לא יוצרים עצמים מטיפוס מחלקה.

תרגילים

1. יש להוסיף למחלקה string המופיעה בגוף הפרק את האופרטור הבא:

```
int string::operator<(string str)
```

האופרטור יחזיר 1 אם המחרוזת str גדולה על פי סדר לקסיקוגרפי, ממחרוזת המחלקה. אחרת - הוא יחזיר 0.

2. כיתבו מחלקה המייצגת שבר פשוט על ידי שני שלמים: מונה ומכנה. כיתבו פונקציית בנאי המקבלת ערכים למונה ולמכנה. כיתבו פונקציות להדפסת השבר, הכפלת שני שברים ואופרטור = של שברים פשוטים. הפעילו את שגרת main הבאה המשתמשת במחלקה:

```
void main(void)
  {
      fraction f1(1,2), f2(1,3), result;
      result=f1*f2;
      result.print();
  }
```

3. כיתבו מחלקה המייצגת נקודה על פי הקואורדינטות (x,y) של הנקודה במישור. יש לספק למחלקה פונקציות העמסה לאופרטורים הבאים:

```
point point::operator ++ (void)
point point::operator = (point p)
point point::operator + (point p)
void  point::point(float x=0, float y=0)
```

***הערה**: הפונקציה ++ מגדילה ב-1 את רכיב x ואת רכיב y של הנקודה.*

4. כיתבו מחלקה המנהלת מילון בן 50 מילים לכל היותר. המילון יכלול שלוש שגרות:

- פונקציית בנייה.
- פונקציית הוספה של מחרוזת למילון, אשר מאפשרת להוסיף את אותה המחרוזת מספר פעמים.
- העמסת האופרטור הבא המחזיר 1 אם המילה s נמצאת במילון, ואחרת - 0:

```
int operator[](char* s)
```

פרק 25

פונקציות סטטיות ומשתני המחלקה

עד השלב הנוכחי, כל עצם שיצרנו בתוכנית, קיבל את סדרת המשתנים הפרטיים שלו. ייתכן מצב שבו נרצה שאלמנטים מסוימים במחלקה יהיו משותפים לכל העצמים מטיפוס המחלקה, על פי דרישות התוכנית והגדרת המשימות לביצוע. לשם כך ניתן להגדיר את האלמנטים המשותפים לעצמים על ידי המילה השמורה static.

בהמשך הפרק נדון בנושאים הבאים:

- ++C מתירה שיתוף של משתנה אחד או יותר, על ידי עצמים מאותו טיפוס מחלקה.
- כאשר מקצים ערך למשתנה משותף, הוא נגיש לכל העצמים מאותו טיפוס מחלקה.
- כדי ליצור משתנה מחלקה משותף, יש לכתוב את המילה השמורה **static** לפני שם המשתנה.
- לאחר שמגדירים איבר במחלקה כסטטי, יש להצהיר על משתנה גלובלי (מחוץ להגדרת המחלקה), אשר מתאים למשתנה המשותף.
- ניתן להשתמש במילה השמורה static כדי לאפשר קריאה של שיטה (פונקציה) במחלקה, עוד לפני שהוגדר עצם כלשהו מטיפוס המחלקה.

שיתוף משתני המחלקה

בדרך כלל, כל עצם מטיפוס מחלקה מסוימת אשר נוצר בתוכנית, מקבל קבוצה של משתני מחלקה לעצמו. אף על פי כן, ייתכן מצב שנרצה שמשתנים מסוימים במחלקה יהיו משותפים לכל העצמים מטיפוס המחלקה. במקרים אלה ניתן להגדיר את המשתנים גם בחלק הפרטי וגם בחלק הציבורי של המחלקה כמשתנים משותפים, על ידי הצבת המילה static לפני שם המשתנה. נעשה זאת כך:

```
private:
  static int shared_value;
```

לאחר הגדרת המחלקה, חייבים להגדיר את המשתנים המשותפים כמשתנים גלובליים מחוץ למחלקה, כמו בדוגמה זו:

```
int class_name::shared_value;
```

בתוכנית **SHARE_IT.CPP** מוגדרת מחלקה בשם book_series. במחלקה זו מוגדר המשתנה page_count כמשתנה משותף לכל העצמים מטיפוס המחלקה הזו. כל שינוי בערך המשתנה ישתקף מייד בכל העצמים מטיפוס המחלקה הקיימים בתוכנית ברגע נתון.

```
#include <iostream.h>
#include <string.h>

class book_series {
 public:
   book_series(char *, char *, float);
   void show_book(void);
   void set_pages(int);
 private:
   static int page_count;
   char title[64];
   char author[64];
   float price;
};

int book_series::page_count;

void book_series::set_pages(int pages)
 {
   page_count = pages;
 }
```

```
book_series::book_series(char *title, char *author, float price)
 {
   strcpy(book_series::title, title);
   strcpy(book_series::author, author);
   book_series::price = price;
 }

void book_series::show_book(void)
 {
   cout << "Title: " << title << endl;
   cout << "Author: " << author << endl;
   cout << "Price: " << price << endl;
   cout << "Pages: " << page_count << endl;
 }

void main(void)
 {
  book_series programming("Rescued by C++" Second
                             Edition", "Jamsa", 22.95);
  book_series word("Rescued by Word for Windows", "Wyatt",19.95);

   word.set_pages(256);

   programming.show_book();
   word.show_book();

   cout << endl << "Changing page count " << endl;

   programming.set_pages(512);

   programming.show_book();
   word.show_book();
 }
```

ניתן לראות שהמשתנה page_count שמוגדר במחלקה הוא מטיפוס static int. מייד אחרי הגדרת המחלקה מוגדר משתנה המחלקה page_count כמשתנה גלובלי בתוכנית. ברגע שנעשה שינוי בערך המשתנה page_count, כל העצמים מטיפוס המחלקה book_series הקיימים בתוכנית מתעדכנים גם הם.

שיתוף משתנים של המחלקה

במצבים מסוימים דרוש שמשתנים מסוימים במחלקה יהיו משותפים לכל העצמים מטיפוס המחלקה. לצורך זה יש להגדיר את המשתנים האלה כ-static. בהמשך חייבים להגדיר את המשתנים המשותפים האלה מחוץ למחלקה כמשתנים גלובליים בתוכנית. כל שינוי בערך המשתנה ישתקף מייד בכל העצמים מטיפוס המחלקה הקיימים בתוכנית.

השימוש במשתנים ציבוריים סטטיים טרם יצירתו של עצם המחלקה

בפרק זה למדנו שהגדרת משתנה המחלקה על ידי שימוש במילה static גורם לכך שהוא יהיה משותף לכל העצמים מטיפוס המחלקה. אולם לפעמים, במהלך התוכנית, צריך להשתמש בעצם כלשהו עוד לפני שהוגדר. כדי לאפשר לתוכנית להשתמש במשתנה משותף, הוא צריך להיות מוגדר בחלק הציבורי של המחלקה. למשל, בתוכנית **USE_MBR.CPP** משתמשים במשתנה page_count, שהוא משתנה של המחלקה book_series, לפני שנוצר עצם מטיפוס המחלקה הזו בתוכנית.

```
#include <iostream.h>
#include <string.h>

class book_series {
 public:
   static int page_count;
 private:
   char title[64];
   char author[64];
   float price;
};

int book_series::page_count;

void main(void)
 {
   book_series::page_count = 256;

   cout << "The current page count is " <<
     book_series::page_count << endl;
 }
```

פונקציות מחלקה סטטיות

בתוכניות הקודמות שבפרק זה הצגנו את השימוש במילה השמורה static עבור משתני המחלקה. בדרך דומה, שפת C++ מאפשרת שימוש במילה static להגדרת פונקציות מחלקה. כאשר מגדירים **פונקציה סטטית משותפת** (**Static member function**), או בשם אחר **פונקציית מחלקה סטטית**, אפשר לקרוא לה מכל מקום בתוכנית, גם אם לא נוצרו כל עצמים שהם. הגדרת פונקציית מחלקה סטטית מאפשרת לה לטפל בנתונים המוגדרים מחוץ למחלקה. למשל, המחלקה menu המתוארת בהמשך נעזרת בדרייבר ANSI למחיקת הנתונים המוצגים על המסך. במערכות שבהן מותקן הקובץ ansi.sys ניתן להשתמש בפונקציית המחלקה clear_screen כדי למחוק את המסך. מכיון שפונקציית המחלקה מוגדרת static, ניתן להפעיל אותה ללא יצירת עצמים מטיפוס menu לפני כן. התוכנית **CLR_SCR.CPP** מפעילה את הפונקציה clear_screen לניקוי המסך.

```
#include <iostream.h>

class menu {
 public:
   static void clear_screen(void);
   // Other methods would be here
 private:
   int number_of_menu_options;
};

void menu::clear_screen(void)
 {
   cout << '\033' << "[2J";
 }

void main(void)
 {
   menu::clear_screen();
 }
```

מכיון שפונקציית המחלקה clear_screen מוגדרת static, ניתן להשתמש בה כדי למחוק את המסך אפילו בהיעדר עצם מטיפוס המחלקה menu. הפונקציה clear_screen משתמשת בתווי הבקרה Esc[2] של דרייבר ANSI למחיקת המסך. מידע נוסף על דרייבר ANSI מצוי בספר "**המדריך השלם לשפת C**" שבהוצאת הוד-עמי.

שימוש בשיטות מחלקה (פונקציות) ישירות בתוכנית

כאשר יוצרים שיטות מחלקה (פונקציות), קורה שפונקציה אשר נוצרה לשימושה של מחלקה מסוימת, יכולה לשמש את התוכנית גם בפעולות שאינן כוללות שימוש בעצמי המחלקה. לדוגמה, בתוכנית הקודמת הוגדרה הפונקציה clear_screen עבור המחלקה menu, אך ניתן להתשמש בה במקומות נוספים במהלך התוכנית. כאשר מחלקה מכילה שיטה שאנו רוצים להשתמש בה מחוץ לעצם המחלקה שבו היא הוגדרה, יש לכתוב לפני אב הטיפוס של השיטה את המילה השמורה static ולהצהיר על השיטה כציבורית:

```
public:
    static void clear_screen(void);
```

במהלך התוכנית נשתמש באופרטור טווח ההכרה כדי לקרוא לפונקציה:

```
menu::clear_screen();
```

סיכום

בפרק זה למדנו שהצבת המילה **static** לפני שם של משתנה במחלקה מאפשר גישה אליו מכל מקום בתוכנית. השימוש במילה שמורה static הופך אותו **למשותף** (**Shared**) לכל העצמים מטיפוס המחלקה הקיימים בתוכנית.

לפני שנעבור לפרק הבא, נבדוק אם מובנים הנושאים הבאים:

- ✓ משתנה מחלקה שמוגדר static, הופך להיות משותף לכל העצמים מטיפוס המחלקה הזו.
- ✓ לאחר שהתוכנית מגדירה משתנה מחלקה כ- static, היא חייבת להגדיר משתנה גלובלי מחוץ להגדרת המחלקה, שמתייחסת למשתנה המשותף.
- ✓ כאשר מגדירים משתנה מחלקה בתור public ו-static, ניתן להשתמש בו גם אם לא הוגדרו עדיין עצמים מטיפוס המחלקה הזו. כדי לגשת למשתנה כזה, התוכנית צריכה להשתמש באופרטור טווח ההכרה הכללי (::), כמו למשל: class_name::member_name.
- ✓ פונקציית המחלקה שנמצאת בחלק הציבורי של המחלקה ומוגדרת static ניתנת לקריאה על ידי התוכנית, עוד **לפני** שהוגדרו עצמים מטיפוס המחלקה הזו. כדי לקרוא לפונקציה, התוכנית צריכה להשתמש באופרטור טווח ההכרה הכללי, כמו למשל: ()menu::clear_screen.

בפרק 26 נציג את המונח **הורשה** (**Inheritance**) ונראה כיצד ניתן להיעזר בהורשה לבניית עצם חדש מתוך עצמים מטיפוסים אחרים. יישום ההורשה ליצירת עצמים חדשים מאפשר חיסכון עצום בשלב התכנות של יישומי מחשב.

תרגילים

1. מהו הפלט של התוכנית הבאה? ענו על שאלה זו לאחר שהפכתם את השורה השגויה (יש אחת כזו בפונקציה main) להערה.

```
#include <iostream.h>
#include <iomanip.h>

class Object {
  public:
    Object(void) { x = y = z = w = 0; }
    int x;
    static int y;
    void print(void);
    static void output_message(void);
  private:
    int w;
   static int z;
 };
int Object::y;
int Object::z;

void Object::print(void)
  {
      cout << x << " " << y << " " << w << " " << z <<     endl;
  }

void Object::output_message(void)
  {
    cout << "This program demonstrates the use of static
                   methods and data members" << endl;
  }

void main(void)
  {
      Object::output_message();
      Object::y = 9;
      Object::z = 17;
      Object s1, s2;
      s1.x = 5;      s1.print();
      s2.x = 7;      s2.y = 3;    s2.print();
      s1.print();
      Object s3;
      s1.print();    s2.print();    s1.y = 4;
      s3.print();
 }
```

חלק 5

הורשה ותבניות

היתרון הגדול ביותר של **תכנות מונחה עצמים** (**Object-Oriented Programming**) הוא יכולת השימוש החוזר בהגדרת המחלקה במספר רב של תוכניות. בחלק זה של הספר נראה ששפת C++ מאפשרת שימוש חוזר של מחלקות בתוכניות רבות, ובנוסף לבנות מחלקות חדשות המתבססות על מחלקות קיימות. כאשר בונים מחלקה חדשה לפי מחלקה שקיימת, מתארים את תהליך הבנייה **כהורשת** תכונות של המחלקה המקורית למחלקה החדשה. בחלק הזה נראה כיצד ניתן לחסוך זמן רב של תכנות על ידי ניצול העוצמה הגלומה בהורשה. עם סיום חלק זה של הספר נוכל לומר שלמדנו חלק ניכר של תכנות בשפת C++ כשפה לתכנות מונחה עצמים.

חלק זה כולל את הפרקים הבאים:

פרק 26

הורשה

אחת המטרות של תכנות מונחה עצמים היא לעשות **שימוש חוזר** (**Reuse**) בחלקים, ולמעשה - מחלקות, שהוכנו עבור תוכנית אחת בתוך תוכנית אחרת. כך רוצים, למשל, שמחלקה שהוגדרה בתוכנית אחת תהיה ניתנת לשימוש חוזר במספר רב של תוכניות שונות, כדי לחסוך בזמן התכנות. כאשר מגדירים מחלקה חדשה, לפעמים היא משתמשת בתכונות של מחלקה קיימת ועליהן מתווספים מספר אלמנטים חדשים ובתוכם משתנים ופונקציות. במצבים אלה שפת C++ מאפשרת לבנות עצמים חדשים המתבססים על תכונות של עצמים קיימים. במילים אחרות, העצם החדש **יורש** (**Inherits**) את האלמנטים של מחלקה קיימת, הנקראת **מחלקה בסיסית** (**Base class**). כאשר בונים מחלקה חדשה מתוך מחלקה קיימת, המחלקה החדשה תיקרא **מחלקה נגזרת** (**Derived class**). בפרק זה נסביר את המושג **הורשה** (**Inheritance**) בשפת C++.

בהמשך הפרק נדון בנושאים הבאים:

- ניצול מחלקה בסיסית לגזירת מחלקה חדשה - זוהי ההורשה.
- קריאה לפונקציות בנייה של המחלקה הבסיסית והמחלקה הנגזרת כדי לאתחל איברי מחלקה.
- גישה לאלמנטים של המחלקה הבסיסית ושל המחלקה הנגזרת באמצעות אופרטור הנקודה.
- האלמנטים המוגנים במחלקה, אשר ניתנים לגישה מהמחלקה הבסיסית ומהמחלקה הנגזרת.
- שימוש באופרטור טווח ההכרה לפתרון ההתנגשויות בין שמות האלמנטים של המחלקה הבסיסית ובין שמות האלמנטים של המחלקה הנגזרת.

הורשה או **ירושה**, כפי שמקובל לעיתים לומר, הוא מושג יסודי בשיטת התכנות מונחה עצמים. לכן מומלץ לשנן היטב את התהליכים המתוארים בתוכניות המוצגות בפרק זה. בפרק זה נראה שההורשה היא שיטה קלה ליישום, וגלומה בה עוצמה רבה המאפשרת לחסוך זמן רב בעבודת התכנות.

הורשה פשוטה

ההורשה (Inheritance) מתבטאת ביכולת של מחלקה בסיסית קיימת להעביר תכונות למחלקה הנגזרת ממנה. המחלקה הבסיסית מורישה תכונות, והמחלקה הנגזרת יורשת תכונות. נניח למשל, ש- employee היא מחלקה בסיסית וזוהי הגדרתה:

```
class employee {
 public:
   employee(char *, char *, float);
   void show_employee(void);
 private:
   char name[64];
   char position[64];
   float salary;
};
```

בהמשך נניח, כי התוכנית זקוקה למחלקה manager המוסיפה לנתוני המחלקה employee את המשתנים הבאים:

```
float annual_bonus;
char company_car[64];
int stock_options;
```

במקרה זה, ניתן לגזור את המחלקה manager מתוך המחלקה הבסיסית employee. כותרת הגדרת המחלקה החדשה מורכבת מהמילה class, שם המחלקה manager, אחריו נקודתיים ושם המחלקה הבסיסית employee.

מחלקה נגזרת

```
class manager : public employee {
    // Members  defined here
};
```

מחלקה בסיסית

המילה public שלפני שם המחלקה employee מציינת שהאלמנטים הציבוריים של המחלקה employee הם אלמנטים ציבוריים גם במחלקה manager.

להלן דוגמה של המשפטים המרכיבים את המחלקה הנגזרת manager:

```
class manager : public employee {
 public:
   manager(char *, char *, char *, float, float, int);
   void show_manager(void);
 private:
   float annual_bonus;
   char company_car[64];
   int stock_options;
};
```

הערה 

הגישה לחלק הפרטי של המחלקה הבסיסית מתוך מחלקה הנגזרת ממנה, יכולה להיעשות באמצעות פונקציות ממשק בלבד. לכן, המחלקה הנגזרת איננה יכולה לגשת בצורה ישירה לאלמנטים הפרטיים של המחלקה הבסיסית תוך שימוש באופרטור הנקודה.

בתוכנית **MGR_EMP.CPP** מתואר יישום של שיטת ההורשה בשפת C++. בתוכנית ישנו תיאור הבנייה של המחלקה manager הנגזרת מהמחלקה הבסיסית employee.

```
#include <iostream.h>
#include <string.h>

class employee {
 public:
  employee(char *, char *, float);
  void show_employee(void);
 private:
  char name[64];
  char position[64];
  float salary;
};

employee::employee(char *name, char *position, float salary)
 {
   strcpy(employee::name, name);
   strcpy(employee::position, position);
   employee::salary = salary;
 }

void employee::show_employee(void)
 {
   cout << "Name: " << name << endl;
```

```
   cout << "Position: " << position << endl;
   cout << "Salary: $" << salary << endl;
 }

class manager : public employee {
 public:
  manager(char *, char *, char *,float, float, int);
  void show_manager(void);
 private:
  float annual_bonus;
  char company_car[64];
  int stock_options;
};

manager::manager(char *name, char *position,
  char *company_car, float salary, float bonus,
  int stock_options) : employee(name, position, salary)
 {
   strcpy(manager::company_car, company_car);
   manager::annual_bonus = bonus;
   manager::stock_options = stock_options;
 }

void manager::show_manager(void)
 {
   show_employee();
   cout << "Company car: " << company_car << endl;
   cout << "Annual bonus: $" << annual_bonus << endl;
   cout << "Stock options: " << stock_options << endl;
 }

void main(void)
 {
   employee worker("John Doe", "Programmer", 35000.0);
   manager boss("Jane Doe", "Vice President", "Lexus",
                                     50000.0, 5000, 1000);

   worker.show_employee();
   boss.show_manager();
 }
```

בתוכנית הקודמת הגדרנו בשלב ראשון את המחלקה הבסיסית employee ורק בהמשך הגדרנו את המחלקה הנגזרת manager. שים לב לפונקציית הבנייה של המחלקה manager. כאשר גוזרים מחלקה חדשה ממחלקה בסיסית, המחלקה הנגזרת חייבת לקרוא לאחת מפונקציות הבנייה של המחלקה הבסיסית. הקריאה לפונקציית הבנייה

של מחלקה בסיסית מתוך המחלקה הנגזרת מתבצעת על ידי רישום שם המחלקה הבסיסית והפרמטרים שלה מייד לאחר הגדרת כותרת הבנאי של הפונקציה הנגזרת. יש להציב את התו נקודתיים בין שם המחלקה הבסיסית לבין כותרת הפונקציה הנגזרת, כמתואר בדוגמה הבאה:

```
manager::manager(char *name, char *position,
  char *company_car, float salary, float bonus,
  int stock_options): employee(name,position,salary)
```

בנאי של מחלקה בסיסית

```
 {
   strcpy(manager::company_car, company_car);
   manager::annual_bonus = bonus;
   manager::stock_options = stock_options;
 }
```

בתוכנית זו, הפונקציה show_manager של המחלקה manager קוראת בצורה ישירה לפונקציה show_employee, שהיא אלמנט המחלקה employee. מכיון שהמחלקה manager נגזרת מהמחלקה employee, הגישה לאלמנטים של המחלקה employee מתבצעת באופן זהה לגישה לאלמנטים השייכים למחלקה הנגזרת עצמה. ובמילים אחרות - כאילו האלמנטים הוגדרו במחלקה manager עצמה.

ההורשה מה היא?

ההורשה היא היכולת של מחלקה נגזרת לקבל בירושה את התכונות של מחלקה בסיסית קיימת. במילים אחרות, זוהי בנייה של מחלקה חדשה על בסיס המשתנים והפונקציות של המחלקה הקיימת (בסיסית). המחלקה החדשה יורשת את האלמנטים (תכונות) של המחלקה הבסיסית. יישום שיטת ההורשה לבניית מחלקות חדשות מאפשר חיסכון משמעותי בזמן תכנות.

דוגמה נוספת

לפנינו הגדרת המחלקה הבסיסית book:

```
class book {
 public:
  book(char *, char *, int);
  void show_book(void);
 private:
  char title[64];
  char author[64];
  int pages;
};
```

נניח כי בתוכנית נדרשת הגדרת המחלקה library_card, הזקוקה לאלמנטים הבאים של המחלקה book.

```
char catalog[64];
int checked_out; // 1 if checked out, otherwise 0
```

במקרה זה ניתן לגזור בדרך זו את המחלקה library_card מתוך המחלקה book:

```
class library_card : public book {
public:
   library_card(char *, char *, int, char *, int);
   void show_card(void);
 private:
   char catalog[64];
   int checked_out;
 };
```

בתוכנית **BOOKCARD.CPP** מתבצעת גזירה של המחלקה library_card מתוך המחלקה book.

```
#include <iostream.h>
#include <string.h>

class book {
 public:
  book(char *, char *, int);
  void show_book(void);
 private:
  char title[64];
  char author[64];
  int pages;
};

book::book(char *title, char *author, int pages)
 {
   strcpy(book::title, title);
   strcpy(book::author, author);
   book::pages = pages;
 }

void book::show_book(void)
 {
   cout << "Title: " << title << endl;
   cout << "Author: " << author << endl;
   cout << "Pages: " << pages << endl;
 }
```

```
class library_card : public book {
 public:
   library_card(char *, char *, int, char *, int);
   void show_card(void);
 private:
   char catalog[64];
   int checked_out;
 };

 library_card::library_card(char *title, char *author,
      int pages, char *catalog, int checked_out) :
      book(title, author, pages)
 {
  strcpy(library_card::catalog, catalog);
  library_card::checked_out = checked_out;
 }

void library_card::show_card(void)
 {
   show_book();
   cout << "Catalog: " << catalog << endl;
   if (checked_out)
     cout << "Status: Checked out" << endl;
   else
     cout << "Status: Available" << endl;
 }

void main(void)
 {
   library_card card("Rescued by C++", "Jamsa", 272,
      "101CPP", 1);

   card.show_card();
 }
```

יש לשים לב לקריאה של בנאי המחלקה book מתוך פונקציית הבנייה של המחלקה הנגזרת library_card. כמו כן, נשים לב לאופן השימוש באלמנט show_book שבמחלקה book בתוך הפונקציה show_card של המחלקה הנגזרת.

האלמנטים המוגנים

בהגדרת המחלקות הבסיסיות ניתן למצוא אלמנטים **פרטיים** (Private), **ציבוריים** (Public) **ומוגנים** (Protected). במחלקה הנגזרת ניתן לגשת לאלמנטים הציבוריים של המחלקה הבסיסית בצורה ישירה, בדומה לאלו המוגדרים בתוך המחלקה הנגזרת. לעומת זאת, אין למחלקה הנגזרת גישה ישירה לאלמנטים הפרטיים של המחלקה הבסיסית. הגישה אליהם מתבצעת באמצעות פונקציות ממשק של המחלקה הבסיסית. תכונות האלמנטים המוגנים במחלקה הבסיסית הם יצור כלאיים בין האלמנטים הפרטיים לבין האלמנטים הציבוריים. המחלקה הנגזרת יכולה לגשת לאלמנט מוגן במחלקה הבסיסית, כמו שהיא ניגשת לאלמנט ציבורי. לעומת זאת, מבחינת שאר חלקי התוכנית האלמנטים המוגנים מתנהגים כאלמנטים פרטיים של המחלקה הבסיסית והגישה אליהם מתבצעת באמצעות פונקציות הממשק בלבד. בהגדרה הבאה של המחלקה book משתמשים בחלק המוגן כדי לאפשר למחלקות הנגזרות ממנה לגשת בצורה ישירה למשתנים title, author ו-page על ידי שימוש באופרטור הנקודה.

```
class book {
 public:
  book(char *, char *, int);
  void show_book(void);
 protected:
  char title[64];
  char author[64];
  int pages;
};
```

השימוש באלמנטים המוגנים מאפשר לחשוב "צעד קדימה" כאשר מגדירים מחלקה. אם נרצה בעתיד לגזור מחלקה חדשה מתוך מחלקה שזה עתה הגדרנו, וגם נרצה לתת למחלקות הנגזרות גישה ישירה לאלמנטים מסוימים - עלינו להגדיר לאלמנטים אלה את התכונה protected (מוגן) ולא private (פרטי).

אלמנטים מוגנים
מספקים נגישות וביטחון

כפי שלמדנו, הגדרת איברי מחלקה כפרטיים - **private**, מונעת מהתוכנית לגשת אליהם ישירות. על כן, כדי לגשת לאיבר פרטי כלשהו, התוכנית צריכה להשתמש ב**פונקציית ממשק** (Interface function). בעת השימוש בירושה, מלאכת התכנות פשוטה יותר, כאשר המחלקות הנגזרות יכולות לגשת לאיברי המחלקה הבסיסית על ידי שימוש באופרטור הנקודה. במקרים כאלה ניתן להשתמש באיברי מחלקה **מוגנים** (Protected). מחלקה נגזרת יכולה לגשת לאיברי מחלקה מוגנים באמצעות אופרטור הנקודה, אך משאר חלקי התוכנית ניתן לגשת אל האיברים המוגנים באמצעות פונקציות הממשק בלבד. בדרך זו, מעמדם של איברי מחלקה **מוגנים** הוא בין איברים

ציבוריים (Public) שאליהם ניתן לגשת מכל חלקי התוכנית, לבין איברים **פרטיים** (Private) שאליהם יכולה לגשת ישירות המחלקה שלהם בלבד.

פתרון להתנגשות שמות אלמנטים במחלקה

במהלך יצירת מחלקות נגזרות עלול להיות מצב, בו שמות אלמנטים של המחלקה הנגזרת זהים לשמות אלמנטים של המחלקה הבסיסית. כאשר יש **התנגשות** שמות, ++C משתמשת באלמנט של המחלקה הנגזרת בתוך הפונקציה של המחלקה הנגזרת. נניח שלכל אחת משתי המחלקות book ו- library_card מוגדר אלמנט בשם price. במחלקה book המשתנה price מקבל את המחיר המומלץ לצרכן, לדוגמה 80 ש״ח (IS). במחלקה library_card המשתנה price הינו המחיר לספריה לאחר הנחה, לדוגמה 55 ש״ח. על פי ההגדרה, מתוך המחלקה library_card ניתן לגשת בצורה ישירה לכל האלמנטים של המחלקה הנגזרת הזו. אולם, אם במחלקה library_card צריך לעדכן את המשתנה price שבמחלקה הבסיסית book, הגישה אליו מתבצעת תוך שימוש באופרטור טווח ההכרה ושם המחלקה הבסיסית (לפי התחביר book::price). אם נרצה להציג על המסך את שני המחירים דרך פונקציית המחלקה הנגזרת show_card, נשתמש במשפטים הבאים:

```
cout << "Library price: $" << price << endl;
cout << "Retail price: $" << book::price << endl;
```

סיכום

בפרק זה למדנו כיצד ניתן לבנות (לגזור) מחלקה חדשה מתוך מחלקה קיימת על ידי יישום תכונות ההורשה (Inheritance) של שפת ++C. לפני שנמשיך, נבחן את עצמנו ונוודא שמובנים לנו הנושאים הבאים:

- ✓ ההורשה היא שיטה המאפשרת למחלקה נגזרת לקבל בירושה את התכונות של מחלקה בסיסית (שממנה היא נגזרת).
- ✓ המחלקה הנגזרת היא המחלקה החדשה, והמחלקה הבסיסית היא המחלקה הקיימת.
- ✓ הגדרת המחלקה הנגזרת מורכבת מהמילה class, שם המחלקה החדשה ושם המחלקה הבסיסית המופרד מהשם הקודם על ידי התו נקודתיים, לפי הדוגמה class dalmatian : dog.
- ✓ במחלקה הנגזרת ניתן לגשת לאלמנטים הציבוריים של המחלקה הבסיסית בצורה ישירה, כאילו הם מוגדרים בתוך המחלקה הנגזרת עצמה. הגישה לאלמנטים הפרטיים של המחלקה הבסיסית מתבצעת באמצעות פונקציות הממשק בלבד.

- ✓ פונקציית הבנייה של מחלקה נגזרת חייבת לקרוא לאחד הבנאים של המחלקה הבסיסית על ידי כתיבת נקודתיים, שם הבנאי של המחלקה הבסיסית והפרמטרים המתאימים, מייד לאחר כותרת פונקציית הבנייה של המחלקה הנגזרת.
- ✓ כדי לאפשר למחלקות הנגזרות לגשת לאלמנטים של המחלקה הבסיסית, אך בו בזמן למנוע משאר מרכיבי התוכנית לגשת אליהם, מגדירים את האלמנטים האלה בחלק המוגן של המחלקה הבסיסית. על ידי כך, המחלקות הנגזרות יכולות לגשת לאלמנטים במחלקה הבסיסית כשם שהן ניגשות לאלמנטים המוגדרים בחלק הציבורי של המחלקה. לעומת זאת, שאר מרכיבי התוכנית רואים אלמנטים אלה כאלמנטים פרטיים.
- ✓ במקרה של שמות זהים לאלמנטים במחלקה הבסיסית ובמחלקה הנגזרת, שפת C++ פונה לשם האלמנט המוגדר במחלקה הנגזרת. הפתרון להתנגשות השמות הוא על ידי שימוש באופרטור טווח ההכרה לפי הדוגמה base_class::member. בדרך זו ניתן לפנות לאלמנט שנמצא במחלקה הבסיסית.

בפרק 27 נלמד כיצד מאפשרת C++ לגזור מחלקה ממחלקה בסיסית אחת, או יותר. הניצול של מספר מחלקות בסיסיות לגזירת מחלקה חדשה אחת הינו תהליך של **הורשה מרובה** (Multiple inheritance).

תרגילים

1. נתונה המחלקה הבאה:

```
class animal {
public:
      animal(char* name);
      void  print(void);
private:
      char  name[80];
};
```

יש לתכנת מחלקה חדשה, dog, היורשת מהמחלקה animal ומוסיפה למחלקה מחרוזת המציינת את גזע הכלב. כיתבו פונקציית בנייה של מחלקת dog, והגדירו מחדש את הפונקציה print (על ידי העמסה - Overloading) להדפסת תכונות הכלב, כולל הגזע. הוסיפו שגרת main לבדיקת המחלקה החדשה.

2. כיתבו מחלקה בשם computer המכילה מחרוזת המתארת את סוג המחשב, ומשתנה המתאר את כמות הזיכרון שלו. כעת יש ליצור שתי מחלקות היורשות מהמחלקה computer ולכל אחת מהן יש מאפיין נוסף: המחלקה pentium היורשת מ-computer ומוסיפה את תכונת מהירות המחשב ביחידות MHz; ומחלקה macintosh, הכוללת משתנה בוליאני (int) המציין אם המחשב הוא מסוג PowerPC. כיתבו פונקציות להשמת ערכים לתכונות, לקבלת ערכי התכונות, וגם להדפסת תכונות המחשבים השונים.

3. כיתבו מחלקה היורשת מהמחלקה library_card (המופיעה גם בגוף הספר).

```
class library_card : public book {
  public:
    library_card(char *, int, char *, int);
    void show_card(void);
  private:
    char catalog[64];
    int checked_out;
  };
```

המחלקה היורשת צריכה להכיל נתונים על ספרים מושאלים: שם השואל, ומספר הימים עד להחזרת הספר. יש להוסיף פונקציה ציבורית להדפסת כל נתוני הספר.

הערה: *מומלץ לבנות פרויקט כדי להריץ את התוכנית. הפרויקט יכיל את הקבצים השונים.*

ex26_3.cpp - הקובץ הראשי המכיל את הפונקציה main. הקבצים האחרים הם:

library_card.h	book.h
library_card.cpp	book.cpp

4. נתונה המחלקה הבאה:

```
class thing
 {
  public:
      thing(char* s)    { strcpy (str, s); }
      void print(void) { cout << str << endl; }

  private:
      char str[80];
 };
```

יש ליצור רשימה מקושרת של עצמים מסוג thing. לשם כך יש ליצור מחלקה בשם thing_list היורשת מהמחלקה thing ומוסיפה משתנה פרטי המצביע לאיבר הבא ברשימה. כיתבו פונקציות של thing_list להוספת איבר לרשימה, ולהדפסת כל הרשימה, וגם פונקציות בנייה ופירוק.

פרק 27

הורשה מרובה

בפרק הקודם למדנו כיצד ניתן לבנות מחלקה חדשה מתוך מחלקה קיימת תוך הורשת תכונות ממחלקה בסיסית למחלקה נגזרת. כעת נרחיב את הנושא ונציג כיצד ניתן לגזור מחלקה מתוך שתי מחלקות בסיסיות או יותר. הורשת התכונות של מספר מחלקות בסיסיות נקרא **הורשה מרובה** (**Multiple inheritance**). בפרק זה נלמד כיצד שפת ++C תומכת בהורשה מרובה.

בהמשך הפרק נדון בנושאים הבאים:

- גזירת מחלקה מתוך מספר מחלקות בסיסיות, הוא תהליך של הורשה מרובה.
- הורשה מרובה גורמת לכך שהמחלקה הנגזרת מקבלת מאפיינים (תכונות) של מחלקה בסיסית אחת, או יותר.
- בתהליך הורשה מרובה, הבנאי של המחלקה הנגזרת חייב לקרוא לפונקציות בנייה עבור כל אחת מהמחלקות הבסיסיות.
- הורשה משורשרת, הינה גזירת מחלקה חדשה מתוך מחלקה נגזרת.

ההורשה המרובה היא כלי **תכנות מונחה-עצמים** (Object-Oriented Programming) בעל עוצמה רבה. מומלץ ללמוד את הנושא על ידי הרצות חוזרות של התוכניות המוצגות בפרק זה. ניתן גם לערוך שינויים קלים וללמוד על השפעתם. יישום ההורשה המרובה חוסך זמן רב בעבודת התכנות.

דוגמה להורשה מרובה

הנה הגדרת המחלקה computer_screen:

```
class computer_screen {
 public:
   computer_screen(char *, long, int, int);
   void show_screen(void);
 private:
   char type[32];
   long colors;
   int x_resolution;
   int y_resolution;
};
```

והנה הגדרת המחלקה mother_board:

```
class mother_board {
 public:
   mother_board(int, int, int);
   void show_mother_board(void);
 private:
   int processor;
   int speed;
   int RAM;
};
```

נוכל להשתמש במחלקות הקודמות לגזירת המחלקה computer.

```
class computer : public computer_screen, public mother_board
{
 public:
   computer(char *, int, float, char *, long, int,
     int, int, int, int);
   void show_computer(void);
 private:
   char name[64];
   int hard_disk;
   float floppy;
};
```

יש לשים לב לכך שבכותרת הגדרת המחלקה הנגזרת מופיעים שני שמות של מחלקות בסיסיות:

```
class computer:
      public computer_screen, public mother_board
```

שתי מחלקות בסיסיות

בתוכנית **COMPUTER.CPP** שלהלן, גוזרים את המחלקה computer מתוך המחלקות computer_screen ו-mother_board.

```
#include <iostream.h>
#include <string.h>

class computer_screen {
 public:
   computer_screen(char *, long, int, int);
   void show_screen(void);
 private:
   char type[32];
   long colors;
   int x_resolution;
   int y_resolution;
};

computer_screen::computer_screen(char *type,
  long colors, int x_res, int y_res)
 {
   strcpy(computer_screen::type, type);
   computer_screen::colors = colors;
   computer_screen::x_resolution = x_res;
   computer_screen::y_resolution = y_res;
 }

void computer_screen::show_screen(void)
 {
   cout << "Screen type: " << type << endl;
   cout << "Colors: " << colors << endl;
   cout << "Resolution: " << x_resolution << "by " <<
                                 y_resolution << endl;
 }

class mother_board {
 public:
   mother_board(int, int, int);
   void show_mother_board(void);
 private:
   int processor;
   int speed;
   int RAM;
};

mother_board::mother_board(int processor, int speed, int RAM)
 {
   mother_board::processor = processor;
   mother_board::speed = speed;
   mother_board::RAM = RAM;
 }
```

```
void mother_board::show_mother_board(void)
 {
   cout << "Processor: " << processor << endl;
   cout << "Speed: " << speed << "Mhz" << endl;
   cout << "RAM: " << RAM << " Mb" << endl;
 }

class computer : public computer_screen,
  public mother_board {
 public:
   computer(char *, int, float, char *, long,
     int, int, int, int, int);
   void show_computer(void);
 private:
   char name[64];
   int hard_disk;
   float floppy;
};

computer::computer(char *name, int hard_disk,
  float floppy, char *screen, long colors,
  int x_res, int y_res, int processor, int speed,
  int RAM) : computer_screen(screen, colors, x_res,
  y_res), mother_board(processor, speed, RAM)
 {
   strcpy(computer::name, name);
   computer::hard_disk = hard_disk;
   computer::floppy = floppy;
 }
void computer::show_computer(void)
 {
   cout << "Type: " << name << endl;
   cout << "Hard disk: " << hard_disk << "Mb" << endl;
   cout << "Floppy disk: " << floppy << "Mb" << endl;
   show_mother_board();
   show_screen();
 }

void main(void)
 {
   computer my_pc("Compaq", 212, 1.44, "SVGA",
     16000000L , 640, 480, 486, 66, 8);

   my_pc.show_computer();
 }
```

יש לשים לב, שמתוך פונקציית הבנייה של המחלקה computer מתבצעת קריאה לשתי פונקציות הבנייה של המחלקות הבסיסיות computer_screen ו-mother_board.

```
computer::computer(char *name, int hard_disk,
  float floppy, char *screen, long colors, int x_res,
  int y_res, int processor, int speed, int RAM) :
  computer_screen(screen, colors, x_res, y_res),
  mother_board(processor, speed, RAM)
```

בניית הורשה משורשרת

כשמשתמשים בשיטת ההורשה בשפת ++C ייתכן מצב שבו נרצה לגזור מחלקה חדשה מתוך מחלקה שהיא עצמה נגזרת ממחלקה אחרת. זוהי **הורשה משורשת** (Inheritance chain). לדוגמה, נרצה להשתמש במחלקה computer כמחלקה בסיסית למחלקה נגזרת work_station.

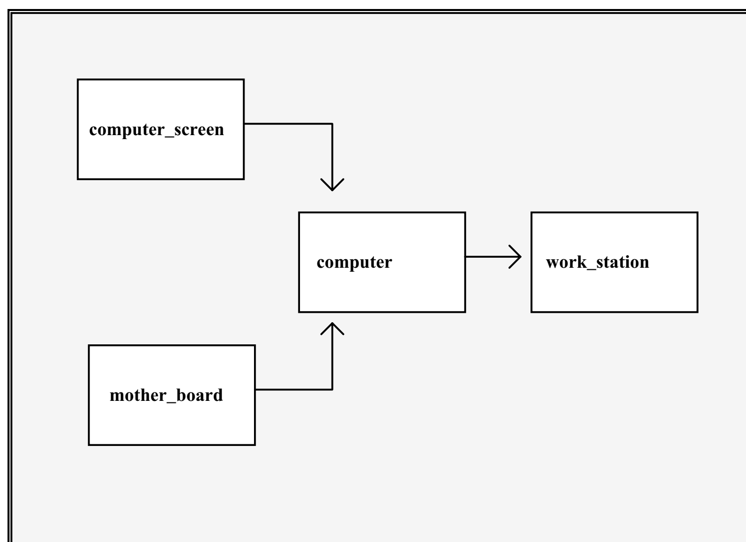
```
class work_station : public computer {
 public:
   work_station(char *operating_system, char *name,
      int hard_disk, float floppy, char *screen,
      long colors, int x_res, int y_res,
      int processor, int speed, int RAM);
   void show_work_station(void);
 private:
   char operating_system[64];
};
```

פונקציית הבנייה של המחלקה work_station קוראת לפונקציית הבנייה של המחלקה computer, אשר בתורה קוראת לפונקציות הבנייה של המחלקות computer_screen ו-main_board.

```
work_station::work_station(char *operating_system,
  char *name, int hard_disk, float floppy,
  char *screen, long colors, int x_res, int y_res,
  int processor, int speed, int RAM) :
  computer (name, hard_disk, floppy, screen,
  colors, x_res, y_res, processor, speed, RAM)
 {
  strcpy(work_station::operating_system,operating_system);
}
```

במקרה זה, המחלקה computer מתפקדת כמחלקה בסיסית, על אף שהיא מחלקה נגזרת מהמחלקות computer_screen ו-main_board. התוצאה הישירה משרשור המחלקות היא, שמחלקת work_station יורשת את תכונותיהן של כל שלוש המחלקות האלו. בתרשים 27.1 מוצגת גזירת המחלקה וההורשה המשורשרת.

ככל שמרבים להשתמש בשיטת ההורשה ובהורשה משורשרת, נראה שמחלקה נגזרת עשויה לקבל מספר רב של אלמנטים שנובעים משרשרת המחלקות שהיא נגזרת מהן.



תרשים 27.1: בניית הורשה משורשרת.

גזירת מחלקה חדשה באמצעות הרשאת private

ב- C++ אפשר להגדיר את המחלקה הנגזרת כך שתיגזר כ- :private באופן הבא:

```
class derive: private base
       {
              גוף המחלקה;
       }
```

כל המשתנים והפונקציות שהוגדרו כ- :protected או כ- :public במחלקת הבסיס, הופכים להיות :private בתוך המחלקה הנגזרת. ולכן, המחלקה הנגזרת תוכל לגשת אליהם, אבל המחלקות שייגזרו ממנה לא יוכלו לגשת למשתנים. מחלקה שנגזרת כ- :private שוברת את שרשרת הגזירה. משתנים שהוגדרו במחלקת הבסיס כ- :protected או כ- :public הופכים ל- :private במחלקה הנגזרת מכיון שזו נגזרה באמצעות הרשאת :private. הפונקציות של המחלקה הנגזרת יכולות לגשת למשתנים שהפכו ל- :private בתוך המחלקה הנגזרת, אבל הן לא יכולות לגשת למשתנים שהוגדרו כ- :private במחלקת הבסיס.

דוגמה : inhrprvt.cpp

```
#include <iostream.h>

class base
  {
    private:
      float x;
    protected:
      float y;
    public:
      float z;
      base(float a, float b, float c){x = a; y = b; z = c;}
      float get_x(){return(x);}
      float get_y(){return(y);}
      float get_z(){return(z);}
      void show(){cout<<"base:\n x= "<<x<<" y= "<<y<<" z=
                                             "<<z<<endl;}
  };

class derive1: private base
  {
    private:
      float x1;
    protected:
      float y1;
    public:
      float z1;
      derive1(float a, float b, float c, float a1, float
                                        b1, float c1)
                                        :base(a, b, c)
        {
          x1 = a1; y1 = b1; z1 = c1;
        }
      float get_x1(){return(x1);}
      float get_y1(){return(y1);}
      float get_z1(){return(z1);}
      void show()
        {
          cout<<"derive1:\n x= "<<get_x()<<" y= "<<y<<" z=
          "<<z<<"\nx1= "<<x1<<" y1= "<<y1<<" z1= "<<z1<<endl;
        }
  };
```

```
class derive2: private derive1
  {
    private:
      float x2;
    protected:
      float y2;
    public:
      float z2;
      derive2(float a, float b, float c, float a1, float b1,
                   float c1,float a2, float b2, float c2)
                        : derive1(a, b, c, a1, b1, c1)
        {
          x2 = a2; y2 = b2; z2 = c2;
        }
      float get_x2(){return(x2);}
      float get_y2(){return(y2);}
      float get_z2(){return(z2);}
      void show()
        {
          cout<<"derive2:\n x1= "<<get_x1()<<" y1= "<<y1<<"z1=
                "<<z1<<"\nx2= "<<get_x2()<<" y2= "<<y2<<" z2=
                "<<z2<<endl;
        }
  };

void main(void)
  {
    derive2 d2(1.0f, 2.0f, 3.0f, 4.0f, 5.0f, 6.0f, 7.0f, 8.0f,
                                                         9.0f);
    d2.show();
    cout<<"main():\n x2="<<d2.get_x2()<<" y2= "<<d2.get_y2()<<
                                  " z2= "<<d2.z2<<endl;
  }
```

סוגי הרשאה והיתרי גישה

גזירה באמצעות public

class		base		derive1		derive2		main()
		הרשאה	גישה	הרשאה	גישה	הרשאה	גישה	גישה
base	x	private	מותרת	private	אסורה	private	אסורה	אסורה
	y	protected	מותרת	protected	מותרת	protected	מותרת	אסורה
	z	public	מותרת	public	מותרת	public	מותרת	מותרת
derive1	x1			private	מותרת	private	אסורה	אסורה
	y1			protected	מותרת	protected	מותרת	אסורה
	z1			public	מותרת	public	מותרת	מותרת
derive2	x2					private	מותרת	אסורה
	y2					protected	מותרת	אסורה
	z2					public	מותרת	מותרת

גזירה באמצעות private

class		base		derive1		derive2		main()
		הרשאה	גישה	הרשאה	גישה	הרשאה	גישה	גישה
base	x	private	מותרת	private	אסורה	private	אסורה	אסורה
	y	protected	מותרת	protected	מותרת	protected	אסורה	אסורה
	z	public	מותרת	public	מותרת	public	אסורה	אסורה
derive1	x1			private	מותרת	private	אסורה	אסורה
	y1			protected	מותרת	protected	מותרת	אסורה
	z1			public	מותרת	public	מותרת	אסורה
derive2	x2					private	מותרת	אסורה
	y2					protected	מותרת	אסורה
	z2					public	מותרת	מותרת

גזירה באמצעות protected

class		base		Derive1		derive2		main()
		הרשאה	גישה	הרשאה	גישה	הרשאה	גישה	גישה
base	x	private	מותרת	private	אסורה	private	אסורה	אסורה
	y	protected	מותרת	protected	מותרת	protected	מותרת	אסורה
	z	public	מותרת	protected	מותרת	protected	מותרת	אסורה
derive1	x1			private	מותרת	private	אסורה	אסורה
	y1			protected	מותרת	protected	מותרת	אסורה
	z1			public	מותרת	protected	מותרת	אסורה
derive2	x2					private	מותרת	אסורה
	y2					protected	מותרת	אסורה
	z2					public	מותרת	מותרת

סיכום סוגי הרשאה והיתרי גישה

base **class A**	**גזירה באמצעות** **private** **class B: private A**	**גזירה באמצעות** **protected** **class B: protected A**	**גזירה באמצעות** **public** **class B: public A**
private:	לא נגיש	לא נגיש	לא נגיש
protected:	private:	protected:	protected:
public:	private:	protected:	public:

תכנון עץ הורשה

הסיבה העיקרית לבניית עץ הורשה היא שימוש בחלקים מהקוד של מחלקת הבסיס במספר מחלקות נגזרות. אובייקט הנגזרת מקבל את המשתנים של מחלקת הבסיס גם אם הוא לא עושה שימוש בכולם. לכן, רצוי ליצור מחלקת בסיס עם מכנה משותף בסיסי ולהוסיף מידע במחלקות הנגזרות. אם ישנם משתנים במחלקת הבסיס שאנו לא עושים בהם שימוש. מומלץ ליצור מחלקת בסיס צרה ובסיסית יותר. רצוי להגדיר פעולות כלליות במחלקת הבסיס כך שלא יידרש בהן שינוי במחלקות הנגזרות. במידה ופונקציות מחלקת הבסיס לא יעבדו בצורה זהה במחלקות הנגזרות, יש להגדירן כ- Virtual. עץ הורשה רחב נוצר כאשר למחלקת הבסיס הרבה מחלקות נגזרות ישירות מהבסיס, כלומר, למחלקות הנגזרות יש מכנה משותף מרכזי אחד. כל שינוי במחלקת הבסיס ישפיע באופן ישיר על מספר רב של מחלקות. יצירת עץ הורשה עמוק מאפשר להגדיר מספר פונקציות במחלקת הבסיס שאינן דורשות הגדרה מחודשת במחלקות הנגזרות. שינוי במחלקת הבסיס ישפיע באופן ישיר על מספר קטן של מחלקות שיכולות לדאוג להמשך תפקוד תקין בהמשך עץ ההורשה. עץ הורשה עמוק מאפשר גמישות תכנותית.

סיכום

הורשה מרובה (**Multiple inheritance**) היא שיטה לגזירת מחלקה מתוך מספר מחלקות בסיסיות. על ידי יישום הורשה מרובה, המחלקה הנגזרת יורשת את התכונות (האלמנטים) של המחלקות הבסיסיות. התמיכה ששפת C++ נותנת להורשה מרובה מוסיפה לתוכניות עוצמה רבה.

לפני שנמשיך לפרק הבא, נבחן אם מובנים הנושאים הבאים :

- ✓ ההורשה המרובה היא היכולת לגזור מחלקה חדשה היורשת את תכונותיהן של מספר מחלקות בסיסיות.
- ✓ כדי לגזור מחלקה חדשה ממספר מחלקות בסיסית יש לציין את שמות המחלקות הבסיסיות ולהפרידן בפסיקים בכותרת הגדרת המחלקה הנגזרת לפי דוגמה זו : class cabbit : public cat, public rabbit.
- ✓ בהגדרת פונקציית הבנייה של המחלקה הנגזרת חייבים לקרוא לפונקציות הבנייה של כל אחת מהמחלקות הבסיסיות, ולהעביר כראוי את הפרמטרים.
- ✓ במהלך גזירת מחלקות חדשות ייתכן מצב בו המחלקה הבסיסית עצמה נגזרת ממחלקות נגזרות אחרות. במקרים אלה נוצרת בתוכנית הורשה משורשרת (Inheritance chain). כאשר פונקציית הבנייה של המחלקה האחרונה בשרשרת קוראת לפונקציית הבנייה של המחלקה הבסיסית לה, שאר פונקציות הבנייה בשרשרת ההורשה מתבצעות לפי התור.

בפרק 28 נציג כיצד ניתן לגשת לאלמנטים הפרטיים של מחלקה באמצעות פונקציות הנמצאות בתוך מחלקה אחרת אשר סווגה כ**חברה** (**Friend**). על ידי שימוש בפונקציות חברות, מאפשרים גישה ישירה של פונקציות מסוימות אל אלמנטים של פונקציות אחרות, בשעה שממשיכים את ההגנה על אותם אלמנטים מגישה על ידי חלקים אחרים של התוכנית.

תרגילים

1. כיתבו תוכנית שתכלול:

א) מחלקה vehicle בעלת איברי מחלקה אלה: weight, speed,volume.

ב) מחלקה for_sale_item בעלת: price ,catalog_nb.

ג) מחלקה car היורשת ממחלקות א׳ ו-ב׳ ומוסיפה איברי מחלקה אלה: nb_wheels ,color.

ד) מחלקה bicycle היורשת ממחלקות א׳ ו-ב׳ ומוסיפה איברים אלה: color, nb_wheels.

כיתבו את הבנאים (constructors) של כל המחלקות הללו.

2. כיתבו תוכנית שתכלול:

א) מחלקה phone בעלת איברי מחלקה : area_code, phone_nb.

ב) מחלקה address בעלת איברי מחלקה: city_name ,street_nb ,street_nam.

ג) מחלקה person_data היורשת ממחלקות א׳ ו-ב׳ ומוסיפה איבר מחלקה: name.

כיתבו את הבנאים (constructors) של כל המחלקות הללו.

הערה: *בשני התרגילים צריך להוסיף את הפונקציה main ואת הפונקציה ()show, כדי להציג את משתני המחלקות.*

פרק 28

אלמנטים פרטיים ואלמנטים חברים

בפרקים הקודמים למדנו שהגישה לאלמנטים פרטיים של המחלקה מתאפשרת באמצעות פונקציות המחלקה בלבד. שימוש רב באלמנטים פרטיים של המחלקה מקטין באופן משמעותי את הסיכוי לשימוש לא נכון באלמנטים של המחלקה, מכיון שהגישה אליהם נעשית באמצעות פונקציות ממשק בלבד. הדרך הזו היא המומלצת ביותר, וטוב לא לסטות ממנה. למרות זאת, לפעמים גישה ישירה למרכיבים מסוימים של התוכנית או לחלק פרטי של מחלקה מסוימת עשויה לשפר באופן משמעותי את ביצועי התוכנית. הגישה הישירה לאלמנטים במחלקה משפרת את זמני התגובה של התוכנית, מכיון שהתוכנית חוסכת את התהליכים הנלווים לקריאת פונקציית הממשק. למקרים אלה, שפת ++C מאפשרת להגדיר מחלקה **חברה** (**Friend**) אשר מותרת לה גישה ישירה ל**אלמנטים הפרטיים** של חברותיה.

בהמשך הפרק נדון בנושאים הבאים:

- ציון קשרי חברות (Friend) בין מחלקות, כדי שמחלקות אחרות תוכלנה לגשת לאיברים הפרטיים (Private) של המחלקה המצהירה על חברותיה.
- איברים פרטיים במחלקה מגינים על נתוני המחלקה - ולכן צריך **להגביל** את השימוש במחלקות חברות רק לאותן מחלקות ש**באמת** צריכות לגשת לאיברי המחלקה הפרטיים.
- הגבלת הגישה של מחלקות חברות לפונקציות מסוימות בלבד.

האלמנטים הפרטיים של המחלקה מגינים עליה ומקטינים את הסיכוי לתקלות ושגיאות. לכן, יש לצמצם את השימוש במחלקות חברות. ככל שמרחיבים את אפשרות הגישה הישירה לאלמנטים של המחלקה, גדל הסיכוי לשגיאות בתוכנית.

הגדרת מחלקה חברה

שפת ++C מאפשרת גישה ישירה לאלמנטים הפרטיים של המחלקה ומחלקות אחרות שהן **חברות** למחלקה נתונה. כדי לציין איזו מחלקה **חברה** (Friend) למחלקה אחרת, יש לרשום בתוך הגדרת המחלקה את שם המחלקה החברה, ולפני השם יש לכתוב את המילה השמורה friend. לדוגמה, בהגדרה הבאה של המחלקה book מציינים שהמחלקה librarian היא מחלקה חברה למחלקה book. כתוצאה מהצהרת החברות הזו, העצמים מטיפוס המחלקה librarian יכולים לגשת בצורה ישירה לאלמנטים הפרטיים של המחלקה book.

```
// viewbook.h
class librarian;

class book {
 public:
   book(char *, char *, char *);
   void show_book(void);
   friend librarian;
 private:
   char title[64];
   char author[64];
   char catalog[64];
};

class librarian {
 public:
   void change_catalog(book *, char *);
   char *get_catalog(book);
};
```

בתוכנית **VIEWBOOK.CPP** נראה שהמחלקה librarian הינה חברה למחלקה book. על כן, לפונקציות של המחלקה librarian יש גישה ישירה לאלמנטים הפרטיים של המחלקה book. בתוכנית זו, הפונקציה change_catalog של המחלקה librarian משנה את המספר הקטלוגי של ספר מסוים. המספר הקטלוגי הוא אלמנט פרטי של המחלקה book.

```
// view book.cpp
#include <iostream.h>
#include <string.h>
#include "viewbook.h"

void main(void)
 {
   book programming("Rescued By C++", "Jamsa", "P101");
   librarian library;
```

```
   programming.show_book();

   library.change_catalog(&programming, "EASY C++ 101");

   programming.show_book();
 }

book::book(char *title, char *author, char *catalog)
 {
   strcpy(book::title, title);
   strcpy(book::author, author);
   strcpy(book::catalog, catalog);
 }

void book::show_book(void)
 {
   cout << "Title: " << title << endl;
   cout << "Author: " << author << endl;
   cout << "Catalog: " << catalog << endl;
 }

void librarian::change_catalog(book *this_book,
      char *new_catalog)
 {
   strcpy(this_book->catalog, new_catalog);
 }

char *librarian::get_catalog(book this_book)
 {
   static char catalog[64];

   strcpy(catalog, this_book.catalog);
   return(catalog);
 }
```

ניתן לראות, שהתוכנית מעבירה את הפונקציה change_catalog של המחלקה librarian אל העצם מטיפוס book על ידי ציון כתובת הפונקציה. חייבים להעביר את הפרמטר על ידי ציון כתובתו, מכיון שהפונקציה משנה תוכן של אלמנט במחלקה. הגישה לאלמנט מתבצעת תוך שימוש במצביע. מומלץ להריץ את התוכנית שנית, אך לפני ההידור וההרצה החוזרת להסיר את המילה friend מתוך הגדרת המחלקה book. כך אנו מסירים מהמחלקה librarian את הגישה הישירה לחלק הפרטי של המחלקה book. המהדר מזהה את המצב ויוצר הודעת שגיאה בכל ניסיון של גישה ישירה לאלמנטים הפרטיים של המחלקה book.

מה הן מחלקות חברות?

בדרך כלל, הדרך היחידה שבה אפשר לגשת לחלק הפרטי של מחלקה היא באמצעות פונקציות הממשק של המחלקה. בהתחשב בנסיבות שונות, יתכן מצב בו יהיה רצוי לאפשר למחלקה מסוימת לגשת ישירות לחלק הפרטי של מחלקה אחרת. אפשרות זו יש לציין למהדר, שמחלקה אחת הינה חברה (Friend) למחלקה אחרת. עושים זאת על ידי רישום המילה friend לפני שם המחלקה החברה ובתוך הגדרת החלק הציבורי של המחלקה שאליה מתאפשרת הגישה הישירה.

```
class abbott {
  public:
    friend costello;
    // Other members
  private:
    // Private members
};
```

השוני בין אלמנטים מוגנים למחלקות חברות

בפרק 26 למדנו, כי אלמנטים מוגנים מאפשרים למחלקות הנגזרות לגשת ישירות אל האלמנטים הפרטיים של מחלקת הבסיס, על ידי שימוש באופרטור הנקודה. יש לזכור, כי רק מחלקות אשר נגזרו ממחלקת הבסיס יכולות לגשת ישירות אל האלמטים המוגנים. במילים אחרות, מחלקות אשר יורשות את האיברים של מחלקת הבסיס. מחלקות חברות (Friend classes) הן בדרך כלל מחלקות שאין ביניהן קשר כלשהו, כלומר, אין ביניהן קשר ירושה. הדרך היחידה בה יכולה מחלקה לגשת אל אלמנטים פרטיים של מחלקה אחרת, שאינה נמצאת איתה בקשר של ירושה, היא במקרה בו המחלקה האחרת מציינת למהדר שהמחלקה הפונה אליה הינה מחלקה חברה (Friend).

הגבלת הגישה של מחלקות חברות

הצהרת חברות מאפשרת למחלקה מסוימת לגשת בצורה ישירה לחלק הפרטי של מחלקה אחרת. ידוע כי פתיחת החלק הפרטי של המחלקה גוררת איתה סיכוי רב יותר לשגיאות ולתקלות בתוכנית. לכן, שפת C++ מאפשרת לקבוע למי מהפונקציות של המחלקה ניתנת הרשות לגשת בצורה ישירה לחלק הפרטי של המחלקה החברה. לדוגמה, נניח שהמחלקה librarian שהצגנו בתוכנית הקודמת מורכבת ממספר רב של פונקציות. בנוסף נניח, שהפונקציות change_catalog ו-get_catalog הן היחידות במחלקה הזו שבהן נדרשת גישה ישירה לנתונים הפרטיים של המחלקה book. בתוך הגדרת המחלקה book ניתן להגביל את הגישה הישירה לחלק הפרטי שלה למחלקות מסוימות בלבד, על ידי ציון הפונקציות החברות האלו בהגדרה. בדוגמה זו הפונקציות הן change_catalog ו-get_catalog.

```
class book {
 public:
  book(char *, char *, char *);
  void show_book(void);
  friend char *librarian::get_catalog(book);
  friend void librarian::change_catalog book *, char *);
 private:
  char title[64];
  char author[64];
  char catalog[64];
};
```

כפי שניתן לראות, המשפטים המתחילים במילה friend כוללים תבניות שלמות של פונקציות עבור כל אחת מפונקציות המחלקה החברות שרשאיות לגשת ישירות אל האיברים הפרטיים של המחלקה.

מה הן פונקציות חברות?

פונקציות חברות (Friend functions) הן דרך להגביל את הגישה של מחלקות חברות אל נתונים בחלק הפרטי של מחלקה אחרת. הגדרת פונקציה חברה בתוך המחלקה כוללת את המילה השמורה friend, ואת האב טיפוס המלא של הפונקציה, כמו בדוגמה זו:

```
public:
friend class_name::function_name(parameter types);
```

רק הפונקציה המסוימת ששמה מפורט במשפט friend יכולה לגשת ישירות לאיברי המחלקה הפרטיים, באמצעות אופרטור הנקודה (Dot).

נוכל לראות שהאלמנטים החברים, המוגדרים עם המילה friend, כוללים את הצהרת הפונקציה החברה במלואה. כאשר יש פניות ממחלקה אחת לחברתה, יכולות להיות שגיאות תחביר אם סדר הגדרת המחלקות אינו נכון. במקרה הקודם, המחלקה book כללה בהגדרתה הצהרות של פונקציות המוגדרות ושייכות למחלקה librarian. לכן, הגדרת המחלקה book חייבת להופיע בתוכנית **לפני** הגדרת המחלקה librarian. אולם הגדרת המחלקה librarian כוללת פנייה למחלקה book, כדלקמן:

```
class librarian {
 public:
   void change_catalog(book *, char *);
   char *get_catalog(book);
};
```

מרקם היחסים בין המחלקה librarian לבין המחלקה book מונע את כתיבת הגדרת המחלקה book לפני הגדרת המחלקה librarian, ולהיפך. שפת ++C מאפשרת פתרון לבעיה על ידי הצבת משפט הגדרה המציין למהדר שההגדרה המלאה של מחלקה נמצאת מאוחר יותר בתוכנית.

הנה המשפט הפותר את הבעיה הקודמת:

```
class book;
```

בתוכנית **LIMITFRI.CPP** מתואר השימוש בפונקציות חברות. בתוכנית זו, פונקציות חברות משמשות להגבלת הגישה של המחלקה librarian אל הנתונים הפרטיים של המחלקה book. יש לשים לב במיוחד לסדר הגדרת המחלקות.

```
#include <iostream.h>
#include <string.h>

class book;

class librarian {
 public:
  void change_catalog(book *, char *);
  char *get_catalog(book);
};

class book {
 public:
  book(char *, char *, char *);
  void show_book(void);
  friend char *librarian::get_catalog(book);
  friend void librarian::change_catalog(book *, char *);
 private:
  char title[64];
  char author[64];
  char catalog[64];
};

book::book(char *title, char *author, char *catalog)
 {
  strcpy(book::title, title);
  strcpy(book::author, author);
  strcpy(book::catalog, catalog);
 }

void book::show_book(void)
 {
   cout << "Title: " << title << endl;
   cout << "Author: " << author << endl;
   cout << "Catalog: " << catalog << endl;
 }
```

```
void librarian::change_catalog(book *this_book, char *new_catalog)
 {
   strcpy(this_book->catalog, new_catalog);
 }

char *librarian::get_catalog(book this_book)
 {
   static char catalog[64];

   strcpy(catalog, this_book.catalog);
   return(catalog);
 }

void main(void)
 {
   book programming("Rescued By C++", "Jamsa", "P101");
   librarian library;

   programming.show_book();

   library.change_catalog(&programming, "EASY C+ 101");

   programming.show_book();
 }
```

מזהה מחלקה

מזהה (Identifier) הוא שם, כמו משתנה או טיפוס מחלקה. כאשר משתמשים במחלקות חברות בתוכנית, יכול להיווצר מצב, בו הגדרת מחלקה מסוימת מתייחסת אל מחלקה אחרת (לפי שם או לפי מזהה), אשר המהדר אינו מכיר עדיין. במקרה כזה, המהדר יודיע על שגיאת תחביר. כדי לפתור את הבעיה הזו, של "איזו מחלקה מוגדרת ראשונה", ניתן לכלול שורה אחת של מזהה מחלקה ליד תחילת קוד המקור:

```
class class_name;
```

המזהה מציין למהדר, כי המחלקה הנוספת תוגדר מאוחר יותר, ולכן התוכנית יכולה להתייחס לשם מחלקה כשם חוקי.

סיכום

בפרק זה למדנו כיצד להשתמש במחלקות חברות (Friend classes) כדי לאפשר שיתוף אלמנטים פרטיים בין מחלקות שונות.

לפני שנעבור לפרק הבא, נבחן אם מובנים הנושאים הבאים:

- ✓ הצהרת חברות מאפשרת למחלקות שונות לגשת באופן ישיר לאלמנטים פרטיים של מחלקות אחרות.
- ✓ כדי להצהיר על מחלקה חברה, יש לציין את המילה השמורה friend, לאחריה את שם המחלקה - כל זה בתוך הגדרת המחלקה.
- ✓ המשמעות של הצהרת מחלקה חברה היא, שכל האלמנטים של אותה מחלקה יכולים לגשת בצורה ישירה לאלמנטים הפרטיים של המחלקה.
- ✓ מגבילים את מספר פונקציות המחלקה המסוגלות לגשת לחלק הפרטי של המחלקה על ידי הגדרת פונקציות חברות. כדי להגדיר פונקציה חברה, כותבים את המילה השמורה friend והצהרה מלאה של הפונקציה, אשר צריכה לפנות לחלק הפרטי של המחלקה.
- ✓ הגדרת פונקציות חברות עלולה לגרום לשגיאות תחביר, בגלל סדר ההגדרה של המחלקות בתוכנית. לפתרון בעיה זו, שפת C++ מאפשרת לציין למהדר שהגדרת המחלקה נמצאת במקום מאוחר יותר בתוכנית, על ידי כתיבת משפט כמו זה: class class_name;.

בפרק 29 נראה כיצד להשתמש **בתבניות פונקציה** (Function templates) כדי לפשט הגדרות של פונקציות דומות.

תרגילים

1. כיתבו שתי מחלקות. הראשונה, המחלקה disk מייצגת דיסק מחשב, ומכילה משתנה מסוג int המתאר את כמות הדיסק הפנוי. משתנה נוסף מתאר את קיבולת האחסון של הדיסק.

   המחלקה השנייה, file, אשר מוגדרת כמחלקה חברה של disk, ומייצגת קובץ על הדיסק. פונקציית הבנייה של file תקבל כפרמטר את גודל הקובץ, ומצביע אל הדיסק, ותעדכן במחלקת הדיסק הרלוונטי את נפח הדיסק הפנוי, בהתאם לגודל הקובץ.

2. התוכנית הבאה מראה את היתרון בפונקציות חברות, המוגדרות כ- friend, על פונקציות המוגדרות במחלקה. נתונה המחלקה הבאה:

```
class complex
  {
    public:
       complex(float real=0, float imagin=0)
                 { re=real; im=imaginary; }
       complex& operator=(const complex& c)
                 { re=c.re; im=c.im; return *this;}

    void print(void);
      private:
        float re,im;
  };
```

השתמש בהגדרות friend כדי לאפשר את הפונקציונליות הבאה:

```
void main(void)
  {
       float f=3;
       complex c(1,2),result;
       result=f+c;
       result=c+f;
       c.print();
       result.print();
  }
```

3. הוסף לתוכנית הקודמת פונקציה חברה המאפשרת הדפסת המספר המרוכב לתוך cout על ידי העמסת האופרטור <<. לשם כך, יש להוסיף פונקציה כזו:

```
ostream& operator<<(ostream& o, complex& c);
```

כעת ניתן להדפיס את המספר על ידי הפקודה הזו:

```
cout << "complex is " << c;
```

בנוסף, בטל את הפונקציה print.

4. לתוכנית **LIMITFRI.CPP** שבגוף הפרק, העוסקת במחלקות book ו-librarian, יש להוסיף טיפול בהשאלת ספרים. צריך ליצור מחלקה בשם booklawn היורשת מהמחלקה book ומוסיפה את שם שואל הספר. הגדר במחלקה librarian את הפונקציה borrow_book לשאילת ספרים, ואת הפונקציה return_book להחזרת ספרים. הגדר פונקציות אלו כחברות במחלקה borrow_book, כדי שאפשר יהיה לגשת למשתנה המכיל את שם שואל הספר.

פרק 29

תבניות פונקציה

במקרים רבים, כאשר מפתחים פונקציות בתוכנית, רואים ששתי פונקציות מבצעות אותה משימה ומתקיים בהן אותו תהליך, והשוני היחיד הוא בטיפוסי המשתנים בהם הפונקציות מטפלות. בפרקים הקודמים למדנו שניתן להעמיס פונקציות וכך להשתמש באותו שם, ולהעביר להן פרמטרים מטיפוסים שונים. אולם, כאשר טיפוס הערך המוחזר שונה, אין מנוס מלהגדיר שתי פונקציות בעלות שמות ייחודיים. נניח שקיימת פונקציה בשם max המחזירה את הערך הגבוה שבין שני ערכים מטיפוס int. אם נצטרך להגדיר פונקציה המבצעת את אותה משימה על ערכים מטיפוס float, נגדיר את אותה התבנית של הפונקציה, אך השם שלה יהיה fmax, למשל. בפרק זה נלמד כיצד להשתמש ב**תבניות** (**Templates**) בשפת ++C, כדי להגדיר בצורה מהירה פונקציות המחזירות ערכים מטיפוסים שונים.

בהמשך הפרק נדון בנושאים הבאים:

- תבניות (Templates) מגדירות קבוצת משפטים שהתוכנית יכולה ליצור מהם פונקציות שונות.
- תוכניות משתמשות לעיתים קרובות בתבניות פונקציה להגדרה מהירה של שתי פונקציות או יותר, אשר מנצלות בדרך כלל את אותה קבוצת משפטים. עם זאת יש להן פרמטרים שונים או ערכי החזרה מטיפוסים שונים.
- לאחר שהתוכנית מגדירה תבנית פונקציה, היא יכולה אחר כך ליצור פונקציה על ידי שימוש בתבנית להגדרת אב טיפוס שיכלול את שם התבנית ואת ערך ההחזר של הפונקציה ואת סוגי הפרמטרים.
- במהלך ההידור של התוכנית, המהדר ייצור בתוכנית את הפונקציות, על פי אבות הטיפוס שמיוחסים לשם התבנית.

התחביר להגדרת התבנית של פונקציה תבנית עלול להיראות, במבט ראשון, מורכב וקשה להבנה. שימוש בתבניות אחדות, ישנה את ההתרשמות הזו במהירות.

הגדרה של תבנית פונקציה פשוטה

הגדרת **תבנית פונקציה** (**Function template**) היא הגדרה נטולת טיפוסי משתנים הקשורים אליה – בשלב מתקדם בתוכנית ייקבעו הטיפוסים לפי הצורך. למשל, ההגדרה הבאה היא הגדרת תבנית לפונקציה max, המחזירה את הערך הגבוה שבין שני הערכים המועברים אליה:

```
template<class T> T max(T a, T b)
 {
   if (a > b)
     return(a);
   else
     return(b);
 }
```

האות T מציינת טיפוס תבנית כללי (Generic). בשלב ההידור, המהדר ממיר את האות T ב**תבנית הפונקציה** בטיפוסי הנתונים הדרושים לכל אחת מהפונקציות המוגדרות. בדוגמה הבאה נראה את היישום float לתבנית הפונקציה max, על ידי המרה של טיפוס המשתנים.

```
 float max(float , float);

            template<class T> T max(T a, T b)
                {
                 if (a > b)
                    return(a);
                 else
                    return(b);
                }
```

בתוכנית הבאה **MAX_TEMP.CPP** מתואר השימוש בתבנית max להגדרת פונקציות מטיפוס float ו-int.

שים ♥ !	**תוכנית זו פועלת בגירסה 5 ומעלה של** Visual C++ **או בגירסה** Borland C++ 3.1 **ומעלה בלבד.**

```
#include <iostream.h>

template<class T> T max(T a, T b)
 {
   if (a > b)
     return(a);
   else
     return(b);
 }
```

```
void main(void)
 {
   cout << "The maximum of 100 and 200 is "
           << max(100, 200) << endl;

   cout << "The maximum of 5.4321 and 1.2345 is "
           << max(5.4321, 1.2345) << endl;
 }
```

השימוש בתבנית הפונקציה

בתוכניות מורכבות יש צורך להגדיר פונקציות דומות המבצעות אותן המשימות, אך משתמשות בטיפוסי ערכים שונים. תבנית הפונקציה מאפשרת להגדיר פונקציה כללית (Generic) או ללא טיפוסי ערכים. כאשר התוכנית צריכה להשתמש בפונקציה מטיפוס מסוים, כמו int או float, כותבים את הצהרת הפונקציה ובה מציינים את שם תבנית הפונקציה. המהדר ממיר את טיפוסי הנתונים ויוצר פונקציות כנדרש. על ידי שימוש בתבנית הפונקציה חוסכים את ההידור החוזר של פונקציות מצד אחד, ומצד שני ניתן להשתמש באותו שם של פונקציה המבצעת את אותן הפעולות. כאשר טיפוס הערך המוחזר וטיפוסי הפרמטרים שונים.

תבניות המטפלות במספר טיפוסים

הגדרת תבנית הפונקציה max הקודמת מתאימה לשימוש בפרמטר מטיפוס מטיפוס T (Generic) בלבד. במקרים רבים צריך לציין מספר טיפוסים שונים. לדוגמה, בהמשך נגדיר תבנית לפונקציה show_array, שתפקידה להציג על המסך תוכן של איברי מערך. בתבנית משתמשים באות T לציון טיפוס המערך, ובאותיות T1 לציון הטיפוס של count (מספר האיברים, במקרה זה).

```
template<class T,class T1> void show_array(T *array, T1 count)
 {
   T1 index;
   for (index = 0; index < count; index++)
   cout << array[index] << ' ';

   cout << end;
 }
```

אפשר, אך אין זו חובה, להגדיר בתוכנית את התבניות של הפונקציות שמתאימות לטיפוסי הפונקציות הדרושים.

הערה



ייתכן והגדרות הפונקציות ייצרו שגיאות בעת הידור עם ++Visual C.

```
void show_array(int *, int);
void show_array(float *, unsigned);
```

בתוכנית **SHOW_TEM.CPP** מתואר השימוש בתבניות פונקציה ליצירת פונקציות המציגות על המסך מערכים מטיפוס int ו- float.

שים ♥ !

תוכנית זו פועלת בגירסה 5 ומעלה של ++Visual C או בגירסה Borland C++ 3.1 ומעלה בלבד.

```
#include <iostream.h>

template<class T,class T1> void show_array (T *array, T1 count)
 {
   T1 index;

   for (index = 0; index < count; index++)
     cout << array[index] << ' ';

   cout << endl;
 }

void show_array(int *, int);
void show_array(float *, unsigned);

void main(void)
 {
   int pages[] = { 100, 200, 300, 400, 500 };
   float prices[] = { 10.05f, 20.10f, 30.15f };

   show_array(pages, 5);
   show_array(prices, 3);
 }
```

תבניות וטיפוסים מרובים

ככל שתבניות פונקציה (Function templates) הולכות ונעשות מורכבות יותר, הן מסוגלות לתמוך בטיפוסים מרובים. למשל, ניתן ליצור תבנית של פונקציה array_sort למיון האיברים של מערך נתון. במקרה זה, עשויה הפונקציה להשתמש בשני פרמטרים, האחד יציין את שם המערך והשני - את מספר האיברים הכלולים בו. אם נצא מנקודת הנחה שבמערך יהיו לא יותר מ- 32,767 ערכים, אזי נוכל להשתמש בטיפוס int עבור הפרמטר של גודל המערך.

עם זאת, תבנית כוללנית יותר תאפשר לתוכנית להגדיר בעצמה את טיפוס הפרמטר, כפי שנראה להלן:

```
template<class T, class T1> void array_sort (T array[],
                                               T1 elements)
  {
      // statements
  }
```

השימוש בתבנית array_sort מאפשר למשל ליצור פונקציה, שתמיין מערך קטן מטיפוס float (שבו פחות מ- 128 איברים), וגם מערך גדול מאוד מטיפוס int, על ידי שימוש באב-טיפוס שונה בכל אחד מהמקרים:

```
void array_sort(float, char);
void array_sort(int, long);
```

סיכום

בפרק זה הראינו שהשימוש בתבניות פונקציה מקל על עבודת התכנות. מהדר שפת C++ יוצר את משפטי הפונקציות הדרושות על פי התבנית, כאשר השוני ביניהן מתבטא בטיפוסי הפרמטרים ובטיפוס הערך המוחזר.

לפני שנמשיך, נבחן אם מובנים הנושאים הבאים:

- ✓ תבנית פונקציה היא הגדרה של פונקציה בעלת טיפוסי נתונים כלליים.
- ✓ כאשר צריכים להשתמש בפונקציה עם טיפוסי נתונים מסוימים, מגדירים בתוכנית תבנית של פונקציה, הכוללת את טיפוסי הנתונים האלה.
- ✓ מהדר שפת C++ ממיר את הטיפוסים הכלליים שבתבנית הפונקציה לפי משפטי הפונקציה שנמצאים בתוכנית, או לפי הקריאה לפונקציה.
- ✓ התוכנית צריכה ליצור תבנית עבור פונקציות נפוצות שפועלות עם טיפוסי נתונים שונים. במילים אחרות, אם דרושה לך פונקציה עם טיפוס נתונים אחד, אינך צריך לבנות תבנית.
- ✓ כאשר הפונקציה זקוקה ליותר מטיפוס נתונים אחד, התבנית מקצה לכל טיפוס מציין יחודי, כמו T, T1, T2. במהלך ההידור, מהדר C++ מקצה לתבנית הפונקציה את הטיפוסים הנכונים שכתבת.

בפרק 30 נלמד להשתמש בתבניות מחלקה, כדי ליצור מחלקות חסרות טיפוס (Typeless), או מחלקות כלליות (Generic).

תרגילים

שים ♥ ! **לפתרון תרגילים אלה השתמשו בגירסה 5 ומעלה של Visual C++ או בגירסה Borland C++ 3.1 ומעלה בלבד.**

1. כיתבו תבנית פונקציה (Function template) המבצעת החלפה (Swap) בין שני ערכים. הכריזו על prototypes של הפונקציה עבור הטיפוסים float ו-int. בידקו את תבנית הפונקציה בעזרת פונקציה main מתאימה.

2. כיתבו תבנית פונקציה המאתחלת את length האיברים הראשונים של מערך מטיפוס T על פי ערך התחלתי מטיפוס T. כיתבו תבנית פונקציה המדפיסה למסך את length האיברים הראשונים של מערך מטיפוס T. בידקו את התוכנית.

3. כיתבו שנית את תרגיל 2, אך הפעם אינדקס המערך הוא מטיפוס כללי G (נומרי).

4. כיתבו תבנית פונקציה הממיינת את length האיברים הראשונים של מערך מטיפוס T בשיטת מיון בועות. כיתבו תבנית פונקציה המדפיסה למסך את length האיברים הראשונים של מערך מטיפוס T. השתמשו בפונקציות אלו כדי למיין מערך של 10 מספרים שלמים.

פרק 30

תבניות מחלקה

בפרק 29 למדנו כיצד להשתמש בתבניות פונקציה להגדרת פונקציות כלליות נטולות ציון מפורש של טיפוסי נתונים המופעלים בהן. כשם שמגדירים פונקציות כלליות להתאמה עתידית לטיפוסי נתונים שונים, כך גם יש צורך לעיתים להגדיר מחלקות כלליות. בשפת ++C ניתן להגדיר **תבניות מחלקה** (**Class templates**).

בהמשך הפרק נדון בנושאים הבאים:

- ניתן ליצור **תבניות מחלקה** (Class template) על ידי שימוש במילה השמורה **template** ובסמלי טיפוס כלליים (Generic), כמו למשל **T**, **T1** ו- **T2**. בהגדרת תבנית המחלקה נוכל להשתמש בטיפוסים כלליים אלה, כדי לציין את הערכים של איברי המחלקה, את הערכים המוחזרים על ידי פונקציות המחלקה, את הטיפוסים של ערכי הפרמטרים ועוד.
- ניתן ליצור עצם מחלקה (Class object) באמצעות תבנית מחלקה, על ידי כתיבת שם המחלקה ואחריה הטיפוסים שהמהדר יקצה לכל אחד מסמלי הטיפוסים הכלליים (יש לכתוב זאת בין סוגריים זוויתיים, כך למשל: <int, float>), ולבסוף יש לכתוב את שם המשתנה.
- כאשר המחלקה מספקת פונקציית בנייה כדי לאתחל את ערכי משתני המחלקה, ניתן לקרוא לפונקציית הבנייה כאשר יוצרים עצם בעזרת התבנית. כך נעשה זאת, לדוגמה: ;class_name<int,float> values(200).
- כאשר המהדר מגלה בתוכנית הצהרה על עצם כזה, הוא יוצר מחלקה על פי התבנית ומשתמש בטיפוסים האקטואליים.

כמו בתבניות פונקציה ניווכח לדעת שאין השימוש בהן מסובך כפי שנראה במבט ראשון וזאת לאחר תרגול עם מספר תבניות מחלקה.

הגדרה של תבנית מחלקה

נגדיר מחלקה הכוללת את הפונקציות הדרושות לחישוב הסכום והממוצע של הערכים המאוחסנים באיברי מערך. במקרה זה המערך הוא מטיפוס int. את המחלקה הזו אפשר להגדיר כך:

```
class array {
 public:
   array(int size);
   long sum(void);
   int average_value(void);
   void show_array(void);
   int add_value(int);
 private:
   int *data;
   int size;
   int index;
 };
```

בתוכנית **I_ARRAY.CPP** מתואר השימוש במחלקה array, עם ערכים מטיפוס int:

```
#include <iostream.h>
#include <stdlib.h>

class array {
 public:
   array(int size);
   long sum(void);
   int average_value(void);
   void show_array(void);
   int add_value(int);
 private:
   int *data;
   int size;
   int index;
 };

array::array(int size)
 {
   data = new int[size];
   if (data == NULL)
    {
     cerr << "Insufficient memory-program ending" << endl;
     exit(1);
    }
```

```
    array::size = size;
    array::index = 0;
  }

long array::sum(void)
  {
    long sum = 0;

    for (int i = 0; i < index; i++)
      sum += data[i];

    return(sum);
  }

int array::average_value(void)
  {
    long sum = 0;

    for (int i = 0; i < index; i++)
      sum += data[i];

    return(sum / index);
  }

void array::show_array(void)
  {

    for (int i = 0; i < index; i++)
      cout << data[i] << ' ';

    cout << endl;
  }

int array::add_value(int value)
  {
    if (index == size)
      return(-1);       // Array is full
    else
     {
       data[index] = value;
       index++;
       return(0);     // Successful
     }
  }
```

```
void main(void)
 {
   array numbers(100);   // 100 element array
   int i;

   for (i = 0; i < 50; i++)
     numbers.add_value(i);

   numbers.show_array();

   cout << "The sum of the numbers is "
                         << numbers.sum() << endl;
   cout << "The average value is "
              << numbers.average_value() << endl;
 }
```

התוכנית מתחילה בהקצאת 100 איברים של המערך. בהמשך מוסיפים למערך 50 איברים על ידי שימוש בפונקציית המחלקה add_value. המשתנה index במחלקה משמש חסם עליון לגודל המערך. אם המשתמש ירצה להוסיף למערך איברים מעבר למוגדר במשתנה זה, הפונקציה add_value של המחלקה תחזיר הודעת שגיאה. המשתנה index משמש בפונקציית המחלקה average_value לחישוב הממוצע של ערכי האיברים במערך. בתוכנית משתמשים באופרטור new להקצאת זיכרון הדרוש לאחסון המערך. נדון באופרטור הזה בפרק 31.

תבניות מחלקה

ככל שתיצור יותר מחלקות, תמצא שמחלקה שהגדרת עבור תוכנית אחת (או עבור התוכנית שאתה עוסק בה כעת), מתאימה גם עבור תוכנית אחרת. לעיתים נמצא שהשוני היחיד בין מחלקות המשמשות בפעולות או בתוכניות שונות, הוא טיפוס איברי המחלקה. לדוגמה, מחלקה התומכת בפעולות על שלמים, לעומת מחלקה דומה התומכת בפעולות על מספרים בנקודה צפה. תבניות מחלקה קיימות, כדי שנוכל לצמצם את כמות התכנות (כלומר, עבודת תכנות, ניפוי ותחזוקה) ולעשות שימוש חוזר בקוד של מחלקות קיימות - על ידי הגדרה מחודשת של מחלקות קיימות. השיטה היא להגדיר תבנית של מחלקה ללא טיפוס (מטיפוס כללי), אשר באמצעותה ניתן יהיה ליצור בעתיד עצמי מחלקה מטיפוסים שונים רצויים. כאשר המהדר מזהה הצהרת עצם המבוססת על תבנית מחלקה, הוא בונה מחלקה חדשה, שבה כל טיפוס כללי מוחלף בטיפוס המתאים לו על פי ההגדרה החדשה. האפשרות ליצור במהירות ובפשטות מחלקות שונות הנבדלות זו מזו בטיפוס בלבד, מפחיתה מאוד את כמות הקוד שצריך לכתוב, ותרומתה הגדולה הינה חיסכון זמן וכסף.

נניח שבתוכנית זו צריך לטפל במערכים מטיפוס int ומטיפוס float. דרך אחת לתמיכה בשתי הטיפוסים היא להגדיר שתי מחלקות שונות, שכל אחת מהן מטפלת בטיפוס מערך אחר. לעומת זאת, הגדרת **תבנית מחלקה** (**Class template**) מונעת כפילות בהגדרת המחלקות. להלן הגדרת תבנית המחלקה array - זוהי תבנית מחלקה כללית.

```
template<class T, class T1>
class array {
 public:
   array(int size);
   T1 sum(void);
   T average_value(void);
   void show_array(void);
   int add_value(T);
 private:
   T *data;
   int size;
   int index;
 };
```

בתבנית מוגדרים שני טיפוסי משתנים כלליים: T ו-T1. במקרה של מערך מטיפוס int, נגדיר את T מטיפוס int ואת T1 מטיפוס long. במקרה של מערך מטיפוס float, נגדיר את שני המשתנים T ו-T1 מטיפוס float. מייד לאחר כל הגדרת פונקציה במחלקה, יש לציין את הטיפוסים הכלליים המתאימים לפונקציה לפי הדוגמה array<T, T1>::average_value. להלן הגדרת פונקציית המחלקה average_value, לפי כללי ההגדרה של תבנית מחלקה.

```
template<class T, class T1>
T array<T, T1>::average_value(void)
 {
   T1 sum = 0;
   int i;
   for (i = 0; i < index; i++)
     sum += data[i];
   return(sum / index);
 }
```

לאחר הגדרת התבנית יוצרים את המחלקות של הטיפוסים הדרושים. עושים זאת על ידי משפט הכולל את שם המחלקה של התבנית וציון טיפוסי הנתונים בין סימני "קטן מ..." ו-"גדול מ...", כמו בדוגמה הבאה:

שם התבנית

```
array<int, long>  numbers(100);
```

טיפוסי התבנית

```
array<float, float>  values(200);
```

עכשיו נראה כיצד התוכנית **GENARRAY.CPP** משתמשת בתבנית המחלקה array.

שים ♥! תוכנית זו פועלת בגירסה 5 ומעלה של Visual C++ או בגירסה Borland C++ 3.1 ומעלה בלבד.

```
#include <iostream.h>
#include <stdlib.h>

template<class T, class T1>
class array {
 public:
   array(int size);
   T1 sum(void);
   T average_value(void);
   void show_array(void);
   int add_value(T);
 private:
   T *data;
   int size;
   int index;
 };

template<class T, class T1>
array<T, T1>::array(int size)
 {
   data = new T[size];

   if (data == NULL)
    {
      cerr << "Insufficient memory-program ending" <<
endl;
      exit(1);
    }

   array::size = size;
   array::index = 0;
 }

template<class T, class T1>
T1 array<T, T1>::sum(void)
 {
   T1 sum = 0;
   for (int i = 0; i < index; i++)
     sum += data[i];
   return(sum);
 }
```

```
template<class T, class T1>
T array<T, T1>::average_value(void)
 {
   T1 sum = 0;

   for (int i = 0; i < index; i++)
     sum += data[i];

   return(sum / index);
 }

template<class T, class T1>
void array<T, T1>::show_array(void)
 {

  for (int i = 0; i < index; i++)
     cout << data[i] << ' ';

   cout << endl;
 }

template<class T, class T1>
int array<T, T1>::add_value(T value)
 {
   if (index == size)
     return(-1);          // Array is full
   else
    {
      data[index] = value;
      index++;
      return(0);          // Successful
    }
 }

void main(void)
 {
   // 100 element array
   array<int, long> numbers(100);
   // 200 element array
   array<float, float> values(200);
   int i;

   for (i = 0; i < 50; i++)
     numbers.add_value(i);
```

```
   numbers.show_array();

   cout << "The sum of the numbers is "
                         << numbers.sum() << endl;
   cout << "The average value is "
          << numbers.average_value() << endl;

   for (i = 0; i < 100; i++)
     values.add_value(i * 100.0);

   values.show_array();

   cout << "The sum of the numbers is "
                        << values.sum() << endl;
   cout << "The average value is "
                        << values.average_value() << endl;
 }
```

הדרך הטובה ביותר להבנת השימוש בתבנית מחלקה היא ליצור שני עותקים של התוכנית הקודמת, כאשר בקובץ המקור הראשון יש להחליף את T ו-T1 ב-int וב-long בהתאמה. בקובץ המקור השני יש להחליף גם את T וגם את T1 ב-float.

הצהרה על עצמים בעזרת תבניות מחלקה

כדי ליצור עצמים באמצעות תבניות מחלקה, צריך לציין את שם תבנית המחלקה ולאחריה, בתוך סוגריים זוויתיים, את הטיפוסים האקטואליים המחליפים את הטיפוסים הכלליים T, T1, T2, וכו׳. לאחר מכן יש לציין את שם העצם (המשתנה) ואת ערכי הפרמטרים שיועברו לפונקציית הבנייה של המחלקה, כפי שנראה להלן:

```
template_class_name<type1, type2> object_name
                              (parameter1, parameter2);
```

כאשר המהדר יפגוש בהצהרה כזו בתוכנית, הוא יצור מחלקה המבוססת על הטיפוסים המפורטים בהצהרה. במשפט הבא מודגם שימוש בתבנית המחלקה array ליצירת מערך מטיפוס char, המאחסן 100 איברים:

```
array<char, int> small_numbers(100);
```

בניית רשימה מקושרת באמצעות Templates

יצירת טיפוס כללי נפוצה ביצירה של מבני נתונים המחזיקים סוגים שונים של נתונים באובייקטים שונים. בתוכנית הבאה נבנה טיפוס כללי של רשימה מקושרת, המאפשר להחזיק אובייקטים מסוגים שונים. נחזיק שתי רשימות מקושרות כל אחת עבור טיפוס שונה של אובייקטים. עבור טיפוס חדש של אובייקטים תיווצר רשימה נוספת ולכן קובץ ה- exe יגדל. יצירת טיפוס כללי של רשימה מקושרת באמצעות פולימורפיזם תיצור קובץ exe קטן יותר, אך תרוץ לאט יותר בגלל הקישור המאוחר.

דוגמה: tmpltlst.cpp

הערה *: התוכנית נבדקה והורצה ב- Visual C++ 6.0.*

```
#include <iostream.h>
#include <string.h>

class number
  {
    private:
      float f;
    public:
      number *next;
      number(float f1){ next = NULL; f = f1; }
      void add (number *new_block)
      {
        next = new_block;
      }
      void show(){ cout << "f = " << f << "  "; }
  };

class string_
  {
    private:
      char s[100];
    public:
      string_ *next;
      string_(const char *str)
        {
          next = NULL;
          strcpy(s, str);
        }
```

```
      void add(string_ *new_block)
        {
          next = new_block;
        }
      void show(){ cout << "s = " << s << "  "; }
  };

template <class T>
class list
  {
    private:
      T *head;
      T *tail;
    public:
      list(void)
      {
        head = NULL;
        tail = NULL;
      }
      void add(T *new_block);
      void print();
  };

template <class T>
void list<T>::add(T *new_block)
  {
    if (head == NULL)
      head = new_block;
    else
      tail->add(new_block);

    tail = new_block;
  }

template <class T>
void list<T>::print()
  {
    T *tmp;

    for (tmp = head; tmp != NULL; tmp = tmp->next)
      tmp->show();
  }
```

```
void main(void)
  {
    number *np;
    string_ *sp;
    list <number> number_list;
    list <string_> string_list;
    char word[100];
    float fn;

    cout << "Enter a float number and a word:\n";
    for (int i = 0; i < 3; i++)
      {
        cin >> fn >> word;
        np = new number(fn);
        sp = new string_(word);
        number_list.add(np);
        string_list.add(sp);
      }
    number_list.print();
    cout << endl;
    string_list.print();
  }
```

סיכום

בפרק זה למדנו שהשימוש בתבנית מחלקה מונע כפילויות בהגדרת מחלקות דומות המטפלות בטיפוסי נתונים שונים. במבט ראשון, הגדרת תבנית מחלקה עלולה להיראות כעבודה מסובכת, בגלל המורכבות הקשורה בהגדרת המחלקה עצמה. לכן, הדרך הטובה להגדרת תבנית מחלקה היא להתחיל בהגדרה רגילה של המחלקה לטיפול בטיפוס נתון אחד. בהמשך חוזרים על הקוד המוגדר ומחליפים בו את טיפוסי הנתונים בטיפוסים כלליים כגון: T, T1, T2.

לפני שנעבור לפרק הבא נבדוק אם מובנים לנו הנושאים הבאים:

- ✓ שימוש בתבנית מחלקה מונע כפילויות בהגדרת מחלקות דומות המטפלות בטיפוסי נתונים שונים.
- ✓ כותרת ההגדרה של תבנית מחלקה כוללת את המילה השמורה template ואת טיפוסי הנתון הכלליים המתוארים על ידי האותיות T, T1, T2 וכו'.
- ✓ כל פונקציה הכלולה בהגדרת תבנית המחלקה, כוללת את משפט template של כותרת המחלקה. בנוסף, יש לציין את טיפוסי הנתון הכלליים בין סוגריים מחודדים (< >). בין שם הפונקציה לבין הצהרת התבנית יש להציב את אופרטור טווח ההכרה לפי הדוגמה:

```
class_name<T,T1>::function_name
```

✓ כדי ליצור מחלקה באמצעות תבנית, יש לציין את שם המחלקה ואחריו את טיפוסי הנתונים הדרושים, אשר רשומים בין סוגריים מחודדים, לפי הדוגמה:

```
class_name <int, long> object
```

תרגילים

שים ♥! **לפתרון תרגילים אלה השתמשו בגירסה 5 ומעלה של Visual C++ או בגירסה Borland C++ 3.1 ומעלה בלבד.**

1. כיתבו תבנית מחלקה (class template) לטיפול בווקטורים בעלי אורך קבוע. על התוכנית לבצע מספר פעולות בשיטה של העמסת האופרטור []: חיבור וחיסור של וקטורים, הקצאת ערכים לווקטורים וגישה לאיברי הווקטור.

2. כיתבו תבנית מחלקה לניהול רשימות מקושרות של איברים כללים. יש לספק שירותי הוספת איבר לרשימה, מחיקת איבר לפי מספרו הסידורי (דוגמה: מחק את איבר 3 מהרשימה), הדפסת כל האיברים למסך, וגם מחיקה של כל הרשימה.

3. הפוך את תבנית המחלקה שבתרגיל 2 לניהול רשומות מקושרות, לתבנית המחזיקה סוגים שונים של אובייקטים.

   הגדר תבנית מחלקה לניהול רשימה מקושרת ובדוק כיצד היא פועלת על אובייקטים שונים.

4. כיתבו את תבנית המחלקה הבאה, המנהלת עץ בינארי ממוין.

```
template <class T>
class tree
  {
  public:
       tree(void);
       ~tree();
       void add(T t);
       void print(void);
  private:
       tree<T>*     left;
       tree<T>*     right;
       T            item;
  };
```

העץ ממוין באופן הבא:

כל ענף בעץ מכיל מצביעים לענפים משמאלו ומימינו. איברים משמאל לענף בהכרח קטנים מהאיבר, בעוד שאיברים מימין לו, גדולים או שווים לו. הפונקציה print תדפיס את כל ערכי העץ בצורה ממוינת על ידי סריקה רקורסיבית של העץ מימין לשמאל.

חלק 6

נושאים מתקדמים ב- C++

עד כה הצגנו את C++ ונושאים בתכנות מונחה-עצמים, אותם צריך להכיר כדי לכתוב תוכניות בעלות עוצמה. בחלק זה, המסיים את הספר, נרחיב את הידע בשפת C++ ונראה כיצד אפשר להקצות זיכרון במהלך הרצת התוכנית. נלמד גם על פעולות קלט/פלט בקבצים. למרות שהפרקים בהמשך נחשבים נושאים מתקדמים בשפת C++, הקושי בהבנתם לא עולה על רמת הקושי שהצגנו בפרקים הקודמים בספר (מלבד 3 הפרקים האחרונים).

חלק זה כולל את הפרקים הבאים:

- ❑ פרק 39 - ניצול מצבים חריגים של ++C לטיפול בשגיאות
- ❑ פרק 40 - ספריית תבניות סטנדרטיות STL
- ❑ פרק 41 - יצירת פרויקט
- ❑ פרק 42 - הרצת תוכנית ב-++Visual C

פרק 31

מאגר הזיכרון החופשי

כפי שלמדנו, כאשר מוגדר מערך בתוכנית, מהדר ++C מקצה עבורו שטח בזיכרון בגודל המתאים לאחסון הערכים של המערך. במשך הזמן גודל המערך המוגדר עלול להיות קטן לצורכי התוכנית המשתנים. למשל, נניח שבתוכנית מוגדר מערך המתאים לאחסון 100 ציוני תלמידים. אם במשך חיי התוכנית צריך לאחסן מספר גדול יותר של ציונים, נצטרך לשנות את התוכנית ולבצע הידור מחדש. דרך אלטרנטיבית היא לגרום לכך שהתוכנית תקצה בעת ההרצה את המקום הנוסף הדרוש בזיכרון המחשב. לצורך הקצאה דינמית של זיכרון יש לציין בתוכנית את גודל הזיכרון הדרוש. שפת ++C מחזירה לתוכנית מצביע לזיכרון שהוקצה עבור המערך. הזיכרון ששפת ++C מקצה בצורה דינמית שייכת ל**מאגר הזיכרון החופשי** (**Free store**).

בהמשך הפרק נדון בנושאים הבאים:

- האופרטור **new** משמש להקצאת זיכרון במהלך ריצת התוכנית.
- השימוש באופרטור new מאפשר לתוכנית לציין באופן דינמי את כמות הזיכרון הנדרשת לה.
- כאשר הקצאת הזיכרון מצליחה, האופרטור new מחזיר מצביע לתחילת מקטע הזיכרון שהוקצה.
- אם פעולת ההקצאה נכשלה (לא קיים מספיק זיכרון, למשל), האופרטור new מחזיר מצביע NULL (ריק).
- האופרטור **delete** משמש לשחרור זיכרון שהוקצה על ידי האופרטור new.

הקצאה ושחרור דינמי של זיכרון ממאגר הזיכרון החופשי היא תכונה בעלת עוצמה רבה. כפי שנלמד ונתרגל, הקצאת זיכרון קלה מאוד לביצוע, למעשה.

האופרטור new

האופרטור **new** מאפשר לתוכנית להקצות זיכרון תוך כדי הרצה. כדי להקצות זיכרון על ידי האופרטור new, יש לציין את גודל הזיכרון הדרוש בבתים. הנה דוגמה למשפט של הקצאה דינמית של 50 בתים בזיכרון. בעזרת האופרטור new נעשה זאת כך :

```
char *buffer = new char[50];
```

כאשר האופרטור new מצליח במשימת הקצאת הזיכרון, הוא מחזיר מצביע המורה על תחילת השטח שהוקצה. במקרה זה, מכיון שהתוכנית מקצה זיכרון לאחסנת מערך תווים, היא מציבה את המצביע שקיבלה לתוך משתנה שמוגדר כמצביע אל טיפוס char. כאשר האופרטור new אינו יכול להקצות את הזיכרון הדרוש, הוא מחזיר מצביע NULL, אשר מכיל את הערך 0. בכל פעם שהתוכנית מקצה זיכרון באופן דינמי באמצעות האופרטור new, היא **חייבת** לבדוק את הערך המוחזר של new, כדי לוודא שאינו NULL.

חשיבות ההקצאה הדינמית של זיכרון

תוכניות רבות עושות שימוש נרחב במערכים כדי לאחסן ערכים רבים מאותו הטיפוס. כאשר מצהירים על מערך, נוהגים בדרך כלל לקבוע את גודלו (מספר האיברים שיהיו כלולים בו). הבעיה היא בכך שלא תמיד יודעים מראש את המספר המקסימלי של איברים בהם תצטרך התוכנית לטפל בזמן הריצה. וגם אם מספר זה ידוע מראש, אין זה תמיד יעיל להקצות מערך בעל גודל מקסימלי, כאשר לרוב דרוש רק חלק קטן ממנו. כאשר מתברר שהתוכנית זקוקה למערך גדול ממה שהוקצה לה בתחילה (למשל, יש מספר רב יותר של תלמידים בקורס, או שמספר הפריטים הלך וגדל במשך הזמן), צריך להגדיר בה מחדש את גודלי המערכים ולהדר אותה שוב.

הקצאה דינמית של זיכרון עוקפת את המגבלות הללו וחוסכת את הטרחה לתקן את התוכנית ולהדר אותה שוב. ההקצאה מאפשרת לתוכנית לפעול בכל ריצה עם כמות הזיכרון הדרושה לה בפועל, וכשזו אוזלת, להקצות באופן דינמי עוד זיכרון באמצעות האופרטור new.

בתוכנית **USE_NEW.CPP** מתואר השימוש באופרטור new. היא מבצעת הקצאה דינמית של זיכרון למערך בגודל 100 בתים. היא מגדירה מצביע בשם pointer לאותו קטע זיכרון.

```
#include <iostream.h>

void main(void)
 {
   char *pointer = new char[100];

   if (pointer != NULL)
     cout << "Memory successfully allocated" << endl;
   else
     cout << "Error allocating memory" << endl;
 }
```

במסגרת הקצאה דינמית של זיכרון, ייתכן שגודל הזיכרון הדרוש לא יהיה זמין במאגר הזיכרון החופשי של המחשב. האופרטור new לא יכול להקצות זיכרון וכתוצאה מכך הוא יציב את הערך NULL למצביע הספציפי. בתוכנית הקודמת מתבצעת בדיקת קיום תנאי זה, כדי לקבוע אם הקצאת הזיכרון הושלמה בצורה תקינה.

כאשר אופרטור new נכשל הוא מחזיר NULL

כאשר אין די זיכרון במאגר הזיכרון החופשי, האופרטור new אינו יכול לבצע את משימתו. במקרה כזה, האופרטור מקצה למצביע את הערך NULL. על ידי בדיקת ערך המצביע, כפי שמודגם בתוכנית הקודמת, ניתן לקבוע אם הדרישה להקצאת זיכרון אכן נענתה וכמות הזיכרון הדרושה סופקה. המשפט הבא הוא דוגמה לשימוש באופרטור new להקצאת זיכרון למערך בעל 500 ערכים מטיפוס ממשי :

```
float *array = new float [500];
```

כדי לקבוע אם הוקצה הזיכרון הנדרש, התוכנית משווה את ערך המצביע ל- NULL כפי שנראה להלן :

```
if (array != NULL)
   cout << "Successful memory allocation" << endl;
else
   cout << "new did not allocate memory" << endl;
```

בתוכנית הקודמת גודל הזיכרון להקצאה צוין בתוך התוכנית. בתוכנית הבאה, **ASK_MEM.CPP**, גודל הזיכרון הדרוש ניתן על ידי המשתמש תוך כדי הרצת התוכנית - ראה להלן.

```
#include <iostream.h>

void main(void)
 {
   int size;
   char *pointer;

   cout << "Type in the array size, up to 30000: ";
   cin >> size;

   if (size <= 30000)
    {
      pointer = new char[size];

      if (pointer != NULL)
        cout << "Memory successfully allocated" << endl;
      else
        cout << "Unable to allocate memory" << endl;
    }
 }
```

כאשר אין אפשרות להקצות את הזיכרון הדרוש, האופרטור new מציב את הערך NULL למצביע הספציפי. בתוכנית **NOMEMORY.CPP** מתבצעת הקצאה חוזרת של זיכרון במנות של 10,000 בתים, עד לרגע שהזיכרון שבמאגר הזיכרון החופשי אוזל. כאשר ההקצאה מתבצעת בצורה תקינה, התוכנית מציגה הודעה מתאימה על המסך. ברגע שהזיכרון אוזל ולא ניתן לבצע הקצאת זיכרון, התוכנית מציגה הודעה מתאימה על מסך והרצת התוכנית מסתיימת.

```
#include <iostream.h>

void main(void)
 {
   char *pointer;

   do {
     pointer = new char[10000];
     if (pointer != NULL)
       cout << "Allocated 10,000 bytes" << endl;
     else
       cout << "Memory allocation failed" << endl;
   } while (pointer != NULL);
 }
```

הערה



כאשר מריצים את התוכנית הקודמת בסביבת מערכת הפעלה DOS ניתן לראות שהתוכנית תסתיים לאחר הקצאת זיכרון מעבר ל- 64KB. לפי ברירת המחדל, רוב המהדרים בשפת ++C בסביבת מערכת הפעלה DOS פועלים לפי **מודל זיכרון קטן** (**Small Memory Model**), שבו גודל מאגר הזיכרון החופשי הוא 64KB. בסביבת העבודה DOS, שטח הזיכרון הרצוף הגדול ביותר שניתן לגשת אליו במסגרת התוכנית מוגבל ל- 64KB. יש אפשרות להגדיל את מאגר הזיכרון החופשי על ידי שינוי המודל במהדר. הסבר מפורט בעניין זה תמצא בספר **"המדריך השלם לשפת C"** מאת רש-ליכטמן, בהוצאת הוד-עמי.

מאגר הזיכרון החופשי, מה הוא?

כל פעם שמריצים תוכנית, מהדר שפת ++C שומר אזור בזיכרון המחשב הנקרא מאגר הזיכרון החופשי (Free store). באמצעות האופרטור new התוכנית מסוגלת להקצות זיכרון מתוך מאגר הזיכרון החופשי. על ידי ניצול של המאגר הזה, ניתן לשחרר את הגבלות התכנות הכרוכות בקביעת גודל המערך מראש. גודל מאגר הזיכרון החופשי שונה בין מערכות מחשב לפי סביבת העבודה, מערכת ההפעלה ומודל הקצאת הזיכרון של אותה מערכת מחשב.

שחרור שטח זיכרון שאינו דרוש עוד

האופרטור new מאפשר הקצאה דינמית של זיכרון ממאגר הזיכרון החופשי. כאשר הצורך בזיכרון המוקצה חולף, ניתן להשתמש באופרטור **delete** כדי לשחרר אותו. כדי לבצע זאת יש לציין את המצביע לזיכרון המיועד לשחרור, לפי הדוגמה:

```
delete pointer;
```

בתוכנית **DEL_MEM.CPP** מוצג השימוש באופרטור delete, המשחרר את הזיכרון שהוקצה על ידי האופרטור new:

```
#include <iostream.h>
#include <string.h>

void main(void)
 {
   char *pointer = new char[100];

   strcpy(pointer, "Rescued by C++");

   cout << pointer << endl;

   delete pointer;
 }
```

על פי ברירת המחדל, הזיכרון שהוקצה ואינו משוחרר במהלך התוכנית, **משתחרר** בצורה אוטומטית בסיום התוכנית. כאשר משתמשים באופרטור delete לשחרור הזיכרון במהלך התוכנית, יהיה אפשר להקצות את הזיכרון החופשי הזה מייד למטרות אחרות.

ההקצאה והשחרור של קטעי זיכרון במהלך התוכנית, הופך אותן לפעולות **דינמיות** (שלא כמו הפעולות המוגדרות בעת הידור התוכנית, העוסקות בהגדרת משתנים).

דוגמה נוספת

בתוכנית **ALLOCARR.CPP** מתבצעת הקצאת זיכרון המתאים לאחסון מערך בגודל 1000 ערכים מטיפוס int. אחר כך מתבצעת השמה של הערכים 1 עד 1000 למערך זה. לבסוף, ערכים אלה מוצגים על מסך. בשלב מתקדם יותר, התוכנית משחררת זיכרון של המערך הקודם ומקצה זיכרון למערך חדש בגודל 2000 בתים מטיפוס float. אחר כך מתבצעת השמה של הערכים 1.0 עד 2000.0 במערך זה.

```
void main(void)
 {
   int *int_array = new int[1000];
   float *float_array;
   int i;

   if (int_array != NULL)
    {
      for (i = 0; i < 1000; i++)
        int_array[i] = i + 1;

      for (i = 0; i < 1000; i++)
        cout << int_array[i] << ' ';

      delete int_array;
    }

   float_array = new float[2000];

   if (float_array != NULL)
    {
      for (i = 0; i < 2000; i++)
        float_array[i] = (i + 1) * 1.0;

      for (i = 0; i < 2000; i++)
        cout << float_array[i] << ' ';

      delete float_array;
    }
 }
```

ככלל, התוכנית **חייבת** לשחרר זיכרון שאינו דרוש לה עוד. לצורך זה היא משתמשת באופרטור delete.

סיכום

בפרק זה למדנו כיצד להקצות ולשחרר זיכרון מתוך מאגר הזיכרון החופשי, באופן דינמי, במהלך ביצוע התוכנית. עושים זאת על ידי האופרטורים new ו- delete, אשר חוסכים את הצורך להדר תוכניות אשר זקוקות להקצאת זיכרון שונה מדי פעם.

בפרק 32 נראה כיצד אפשר לנהל את מאגר הזיכרון החופשי ואת הקצאות הזיכרון הנעשות מתוכו, ואת הפעולות שאופרטור new עושה כאשר אין לו אפשרות לענות על דרישת זיכרון.

לפני שנעבור לפרק הבא, נבדוק אם מובנים הנושאים הבאים:

- ✓ יכולת הקצאת הזיכרון הדינמית מורידה את התלות של התוכנית במערכים בגודל קבוע.
- ✓ האופרטור new מחזיר כתובת של האזור בזיכרון כאשר הקצאת הזיכרון מצליחה; אחרת, הוא מחזיר את הערך NULL, כלומר 0.
- ✓ בכל פעם שהתוכנית מקצה זיכרון באמצעות new, היא **חייבת** לבדוק את הערך שהוא מחזיר, כדי לוודא שהאופרטור מחזיר ערך של כתובת זיכרון ולא 0 (אין זיכרון פנוי).
- ✓ הגישה לזיכרון המוקצה על ידי אופרטור new מתבצעת על ידי שימוש במצביע, או במערך.
- ✓ האופרטור new מקצה זיכרון מתוך מאגר הזיכרון החופשי של המערכת - free store.
- ✓ גודל מאגר הזיכרון החופשי שונה בין מערכות מחשב כתלות בסביבת העבודה של מערכת ההפעלה ובמודל הקצאת הזיכרון של המהדר. בסביבת עבודה DOS מאגר הזיכרון החופשי מוגבל ל- 64KB.
- ✓ כאשר חדל הצורך בזיכרון המוקצה, **צריך** לשחרר אותו על ידי שימוש באופרטור delete, כדי להחזירו למאגר הזיכרון החופשי, ולאפשר שימוש חוזר שלו כשנדרש.

תרגילים

1. כיתבו תוכנית בה המשתמש נשאל כמה מספרים (מטיפוס int) הוא רוצה להזין. בהתאם לתשובתו, נוצר מערך דינמי (מטיפוס int) בגודל מתאים ואז הוא מתבקש לספק ערך לכל אחד מאיברי המערך. לאחר הזנת הערכים מודפסים איברי המערך, ולאחר מכן, משחררים את שטח הזיכרון שנתפס על ידי המערך.

2. כיתבו מחלקה של מחסנית (stack) דינמית, כזו שגדלה ומצטמצמת באופן דינמי. המחסנית תומכת בפעולות אלו:

push(X) - הוספת X לראש המחסנית.

pop() - סילוק והחזרת הערך מראש המחסנית.

empty() - האם המחסנית ריקה?

print() - מדפיסה את איברי המחסנית, מהעליון (זה שהוזן אחרון) ועד התחתון.

איברי המחסנית הם מטיפוס int.

פרק 32

פעולות במאגר הזיכרון החופשי

בפרק הקודם למדנו כיצד להשתמש באופרטור new ולהקצות זיכרון מתוך מאגר הזיכרון החופשי. כאשר ההקצאה מושלמת, התוכנית מקבלת מצביע המכיל את הכתובת של קטע הזיכרון שהוקצה; אחרת, C++ מחזירה את הערך NULL למצביע זה. למרות זאת, לפעמים מתעורר הצורך לבצע משימות הדורשות זיכרון על אף אי היכולת של מהדר C++ לספק את דרישות הזיכרון לתוכנית. בפרק זה נציג כיצד להנחות את המהדר לבצע משימות מסוימות גם כאשר הבקשה להקצאת זיכרון נדחית.

בהמשך הפרק נדון בנושאים הבאים:

- הגדרת פונקציית מנהל, הנקראת כאשר בקשה להקצאת זיכרון אינה מקבלת את ההקצאה המספקת.
- הגדרת אופרטור new פרטי.
- הגדרת אופרטור delete פרטי.

כפי שתראה בהמשך, על ידי יצירת אופרטורים מסוג new ו-delete המותאמים לצרכיך, תוכל לשלוט בדרך טובה יותר בתוכנית במצבים של חוסר זיכרון, וגם תוכל לדווח ולטפל טוב יותר במצבים אלה.

יצירה של מנהל מאגר הזיכרון החופשי

בפרק 31 למדנו שהאופרטור new מחזיר את הערך NULL לתוכנית, כאשר אין הוא מוצא את המקום הפנוי הדרוש במאגר הזיכרון החופשי, לפי בקשת התוכנית. בתוכנית **USE_FREE.CPP** מתבצעת קריאה חוזרת של האופרטור new. בכל פעם התוכנית מבקשת להקצות 1000 בתים מתוך מאגר הזיכרון החופשי עד שהמאגר מתרוקן.

```
#include <iostream.h>

void main(void)
 {
   char *pointer;

   do {
     pointer = new char[1000];

     if (pointer != NULL)
       cout << "Allocated 1000 bytes" << endl;
     else
       cout << "Free store empty" << endl;

   } while (pointer);
 }
```

הלולאה שבתוכנית מתבצעת עד שמתקבל הערך NULL. ניתן להגדיר פונקציה שתפעל במצב זה, כאשר יש צורך לבצע משימה מסוימת והתוכנית דורשת הקצאת זיכרון שאינה נענית. לדוגמה, הפונקציה end_program מציגה הודעה על המסך ובהמשך היא נעזרת בפונקציית הספריה הסטנדרטית exit, כדי להפסיק את הרצת התוכנית.

```
void end_program(void)
 {
   cout << "Memory request cannot be satisfied" << endl;
   exit(1);
 }
```

כדי לבצע את הפונקציה end_program במצב שבו המצביע מקבל ערך NULL, צריך לקרוא לפונקציה set_new_handler ולציין בה כפרמטר את שם הפונקציה end_program. הנה דוגמה:

```
set_new_handler(end_program);
```

בתוכנית **END_FREE.CPP** מתבצעת הפונקציה end_program כאשר נדחית הדרישה להקצאת זיכרון מתוך מאגר הזיכרון החופשי. חשוב מאוד לרשום את הפונקציה exit(1); בסוף הפונקציה end_program, כדי למנוע לולאה אינסופית.

שים ♥ ! תוכנית זו פועלת בגירסה Borland C++ 3.1 ומעלה בלבד.

```
#include <iostream.h>
#include <stdlib.h>  // Exit prototype
#include <new.h>     // set_new_handler prototype

void end_program(void)
 {
   cout << "Memory request cannot be satisfied" << endl;
   exit(1);
 }

void main(void)
 {
   char *pointer;

   set_new_handler(end_program);

   do {
     pointer = new char[10000];

     cout << "Allocated 10000 bytes" <<endl;

   } while (1);
 }
```

במקרה זה, כאשר אין מספיק זיכרון פנוי להקצאה, הפונקציה גורמת להפסקת התוכנית. בהתאם לדרישות התוכנית ניתן להיעזר בפונקציה ולהקצות זיכרון מתוך משאבי זיכרון נוספים, כמו לדוגמה, **זיכרון מוגדל** (**Extended memory**) בסביבת DOS. אפשרות נוספת היא שחרור זיכרון שהוקצה לטובת משימה אחרת, והמשך הרצת התוכנית כשלרשותה זיכרון פנוי להקצאה.

יצירת אופרטורים new ו-delete פרטיים

אחת התכונות של שפת ++C היא העמסת אופרטורים. לפי תכונה זו ניתן להגדיר מחדש גם את האופרטורים new ו-delete, ולשנות את **התנהגותם** הרגילה. נציג לדוגמה מקרה דימיוני, שבו צריך להקצות 100 בתים בזיכרון לשם אחסון זמני של הסודות הכמוסים של החברה. כאשר משתמשים באופרטור delete לשחרור הזיכרון שהוקצה, הזיכרון משתחרר, אך המידע נשאר בו עד לשימוש החוזר בשטח זיכרון זה ומחיקתו על ידי דריסה בנתונים חדשים. באופן תיאורטי, מרגל תעשייתי בעל ידע במחשבים ותכנות יכול לאתר את שטח הזיכרון הרגיש הזה ולגלות את סודות החברה.

על ידי העמסת האופרטור delete ניתן לקבוע שלפני שהתוכנית המקורית משחררת את הזיכרון, היא ממלאת אותו (דורסת) באפסים, או בתווים אחרים נטולי משמעות. בתוכנית **MYDELETE.CPP** מתוארת העמסת האופרטור delete. הפונקציה דורסת תחילה את מאה הבתים שכתובתם נמצאת במצביע, ורק אחר כך היא משחררת את קטע הזיכרון שהוקצה. את הזיכרון משחררים על ידי שימוש בפונקציית הספריה הסטנדרטית free.

שים ♥! **תוכנית זו פועלת בגירסה 5 ומעלה של ++Visual C או בגירסה ++Borland C 3.1 ומעלה בלבד.**

```
#include <iostream.h>
#include <stdlib.h>
#include <string.h>

static void operator delete(void *pointer)
 {
   char *data = (char *) pointer;
   int i;

   for (i = 0; i < 100; i++)
     data[i] = 0;

   cout << "The secrets are safe!" << endl;

   free(pointer);
 }

void main(void)
 {
   char *pointer = new char[100];

   strcpy(pointer, "My Company Secrets");

   delete pointer;
 }
```

בתחילת התוכנית היא מקצה זיכרון למערך תווים על ידי שימוש באופרטור new. נתוני החברה מועתקים למערך זה. מאוחר יותר, התוכנית משתמשת באופרטור delete החדש (העמסת האופרטור) ומשחררת את הזיכרון בתוך הפונקציה delete. המשפט הבא מבצע השמה של מצביע מערך התווים אל מצביע של מערך התווים.

```
char *data = (char *) pointer;
```

הביטוי (* char) נקרא **מתאם** (**cast**). מטרת המתאם היא לציין למהדר ++C שהפונקציה מודעת להיותה מציבה מצביע מטיפוס void (ראה את נושא הפרמטרים של הפונקציות) למצביע מטיפוס char. אי הצבת המתאם מונעת את ההידור של

התוכנית. בהמשך מועתקים אפסים אל 100 הבתים של הזיכרון המיועד לשחרור. שחרור הזיכרון נעשה על ידי הפונקציה הסטנדרטית free. יש להדגיש שפונקציה זו פועלת על קטע זיכרון שגודלו 100 בתים, ומכיון שבתוכנית מתבצעת הקצאת זיכרון פעם אחת בלבד, הפעולות מתבצעות בצורה תקינה. אם משנים את התוכנית כדי שתקצה 10 בתים בלבד ולא משנים את פונקציית השחרור בהתאם, התוצאה תהיה **דריסה** של 90 בתים בזיכרון, אשר עלולים להיות בשימוש, ולגרום לתקלה רצינית בביצוע התוכנית. השימוש בפונקציות סטנדרטיות לטיפול במאגר הזיכרון החופשי עשוי להביא תועלת מרובה למתכנת ולביצועי התוכנית. אפשר למשל לזהות את קטע הזיכרון שהמצביע מורה עליו וכך להתייחס אליו בלבד בצורה נכונה. ראוי להכיר היטב את פונקציות הספריה הסטנדרטיות.

בתוכנית **NEW_OVER.CPP** מוגדרת העמסת האופרטור new. במקרה זה הפונקציה המועמסת מציבה בתחילת הזיכרון שהוקצה את המחרוזת הזו:

```
Rescued by C++!
```

```
#include <iostream.h>
#include <alloc.h>
#include <string.h>

static void *operator new(size_t size)
 {
   char *pointer;

   pointer = (char *) malloc(size);

   if (size > strlen("Rescued by C++!"));
     strcpy(pointer, "Rescued by C++!");

   return(pointer);
 }

void main(void)
 {
   char *str = new char[100];
   cout << str << endl;
 }
```

הקצאת הזיכרון בפונקציה new מתבצעת על ידי הפונקציה הסטנדרטית malloc. אם גודל הזיכרון המוקצה גדול מאורך המחרוזת "Rescued by C++!" הפונקציה קוראת לפונקציה הסטנדרטית strcpy כדי להעתיק את המחרוזת לתחילת קטע הזיכרון שהוקצה.

סיכום

בתוכניות מורכבות ברור הצורך והתועלת של השימוש באופרטור new להקצאת זיכרון במהלך ריצת התוכנית. בפרק זה למדנו כיצד ניתן לנווט את פעולת האופרטור new, על ידי הקריאה לפונקציה set_new_handler. זו תגרום לקריאה אוטומטית של פונקציה מסוימת, כאשר האופרטור new אינו מסוגל להקצות זיכרון מתוך מאגר הזיכרון החופשי. הצגנו גם דרך אלטרנטיבית לפעולה זו: העמסת האופרטור new עצמו.

לפני שנעבור לפרק הבא, נבחן אם מובנים לנו הנושאים הבאים:

- ✓ כאשר האופרטור new אינו מסוגל להקצות את קטע הזיכרון הדרוש לתוכנית, הוא מחזיר את הערך NULL למצביע המתאים.
- ✓ קיימת האפשרות להורות לתוכנית לבצע סדרה של פעולות מסוימות כאשר האופרטור new אינו מצליח להקצות את הזיכרון הדרוש. על ידי שימוש בפונקציה set_new_handler האופרטור new קורא לפונקציה מוגדרת בתוכנית כאשר הקצאת הזיכרון נכשלת.
- ✓ בשפת ++C ניתן להעמיס את האופרטורים new ו-delete. לפני העמסת האופרטורים האלה, מומלץ ללמוד כראוי את תהליכי הקצאה והשחרור של הזיכרון ולהכיר את הפונקציות הסטנדרטיות המטפלות בנושא.

תרגילים

שים ♥ ! **לפתרון תרגילים אלה השתמשו בגירסה 5 ומעלה של ++Visual C או בגירסה Borland C++ 3.1 ומעלה בלבד.**

1. כיתבו תוכנית המגדירה את האופרטורים new ו-delete ובודקת שאין דליפות זיכרון, על ידי ספירת מספר פעולות new ו-delete שבוצעו והשוואת מספר פעולות new למספר פעולות delete, שבוצעו במהלך ריצת התוכנית.

2. הגדירו פונקציה memory_allocation_error אשר מופעלת כאשר לא ניתן לספק בקשה להקצאת זיכרון. הפונקציה מדפיסה את נפח הזיכרון הכולל שנותר לשימוש. הפונקציה מנצלת את פונקציית הספריה coreleft.

3. נתונה תוכנית המגדירה את המחלקה הבאה:

```
class fast{
      char data[34];
};
```

התוכנית מבצעת מספר רב של פעולות new ו-delete, אך עם זאת, קיימים לכל היותר 50 אובייקטים מסוג fast בו-זמנית. יש לכתוב את האופרטורים new, ו-delete מהירים, שכאשר הם נדרשים לספק בדיוק 34 בתי זיכרון, הם אינם מפעילים malloc, אלא מנהלים הקצאות אובייקטים מסוג fast מתוך מערך בן 50 אובייקטים שהוקצה בתחילת התוכנית.

4. מה עושה קטע התוכנית הבא:

```
class X {
public:
    void* operator new(size_t size) {
          instances++;
          return malloc(size);
    }
static int instances;
};

int X::instances=0;

void main(void)
{
    X* x=new X;
    delete x;
}
```

פרק 33

ניצול מירבי של CIN ו-COUT

במהלך הלימוד והצגת הדוגמאות, השתמשנו בערוץ הפלט ובפקודה cout להצגת הודעות ונתונים על המסך. במקביל, השתמשנו בערוץ הקלט ובפקודה cin לקליטת נתונים מהמקלדת. בפרק זה נציג את הפקודות cin ו-cout כעצמים, המוגדרים בקובץ הכותר iostream.h (header file). על פי הגדרתם כעצמים, cin ו-cout תומכים באופרטורים שונים ובפעולות שונות. בפרק זה נראה כיצד נכיר את הדרכים לשכלל את התכונות של הקלט והפלט תוך כדי שימוש בפונקציות הבנויות לתוך מחלקות של cin ו-cout.

בפרק זה נדון בנושאים הבאים:

- ❖ תוכן קובץ הכותר iostream.h, המכיל הגדרות מחלקה שאפשר לבחון כדי להבין טוב יותר את ערוצי הקלט/פלט.
- ❖ שליטה על הפלט באמצעות הפונקציה cout.width.
- ❖ המרת טאב ותו רווח בתו אחר, על ידי שימוש בפונקציה cout.fill.
- ❖ שליטה בהצגת מספר ספרות עשרוניות של משתנה מטיפוס float באמצעות הפונקציה cout.setprecision.
- ❖ הצגה וקליטה של תווים בודדים על ידי שימוש בפונקציות cout.put ו-cin.get בהתאמה.
- ❖ קליטה של שורה שלמה באמצעות הפונקציה cin.getline.

כמעט כל תוכנית בשפת ++C משתמשת בתכונות וביכולת הקלט/פלט של העצמים cin ו-cout. לכן מומלץ ללמוד נושא זה בקפידה על ידי התנסות בהרצת התוכניות המוצגות בפרק זה.

קובץ הכותר iostream.h

בכל התוכניות המוצגות בספר כלול תמיד קובץ הכותר iostream.h. קובץ זה מאפשר להשתמש בתוכנית ב-cin לביצוע פעולות קלט וב- cout - לביצוע פעולות פלט. קובץ זה מכיל את הגדרת המחלקות **istream** (**ערוץ הקלט**) ו-**ostream** (**ערוץ הפלט**), שבהן cin ו-cout הם עצמים. בגין חשיבות הנושא מומלץ להדפיס את הקובץ וללמוד את תוכנו. הקובץ צריך להיות בתת-הספריה INCLUDE של המהדר. ההגדרות שבקובץ הן די מורכבות, אך לימוד קפדני של הקובץ יגלה שרובו מורכב מהגדרות של מחלקות וקבועים. בתוך הקובץ ניתן למצוא את הגדרת העצמים cin ו-cout.

השימוש ב-cout

המחלקה cout כוללת מספר שיטות או פונקציות שונות. בתוכניות שבהמשך נתאר את השימוש במספר פונקציות המשמשות לעיצוב פלט התוכנית. בפרק 3 למדנו שהמשפט setw מאפשר לקבוע את מספר המקומות (פוזיציות) המינימלי שבו יוצג ערך הפלט על המסך (הכוונה תמיד לערך שאחרי setw).

```
#include <iostream.h>
#include <iomanip.h>

void main(void)
 {
 cout << "My favorite number is" << setw(3) << 1001 << endl;
 cout << "My favorite number is" << setw(4) << 1001 << endl;
 cout << "My favorite number is" << setw(5) << 1001 << endl;
 cout << "My favorite number is" << setw(6) << 1001 << endl;
 }
```

בצורה דומה, הפונקציה cout.width מאפשרת לקבוע את מספר התווים להצגת הפלט של הביטוי שאחרי הפונקציה. בתוכנית **COUTWIDT.CPP** מתואר השימוש בפונקציה זו, כדי לבצע את הפעולות שתארנו.

```
#include <iostream.h>
#include <iomanip.h>

void main(void)
 {
   int i;

   for (i = 3; i < 7; i++)
    {
      cout << "My favorite number is";
      cout.width(i);
      cout << 1001 << endl;
    }
 }
```

לאחר הידור והרצת התוכנית יוצגו על המסך השורות הבאות:

```
C:\> COUTWIDT <Enter>
My favorite number is1001
My favorite number is1001
My favorite number is 1001
My favorite number is  1001
```

להזכיר, בדומה למשפט setw, גם הפונקציה cout.width משפיעה על ביטוי הפלט הסמוך לה.

תווי מילוי

השימוש במשפט setw או בפונקציה cout.width גורם להצבת מספר תווי רווח לפני או אחרי ערכי הפלט המטופלים בהם, כנדרש. ייתכן מצב שנרצה להשתמש בתו שונה מתו הרווח, כי כך הן דרישות התוכנית. למשל, נניח שהתוכנית יוצרת את הטבלה שלהלן:

```
Table of Information

Company Profile................................10

Company Profit and Loss........................11

Company Board Members..........................13
```

בטבלה זו מופיעות נקודות לפני מספר הדף, הנקראות **נקודות מובילות** (**Dot leaders**). הנקודה הינה במקרה זה **תו מילוי** (**Fill character**). באמצעות הפונקציה cout.fill ניתן להחליף את תווי הרווח בתו כלשהו אחר.

בתוכנית **COUTFILL.CPP** עורכים טבלה דומה.

```
#include <iostream.h>
#include <iomanip.h>

void main(void)
 {
   cout << "Table of Information" << endl;
   cout.fill('.');
   cout << "Company Profile" << setw(20) << 10 << endl;
   cout << "Company Profit and Loss" << setw(12) << 11 << endl;
   cout << "Company Board Members" << setw(14) << 13 << endl;
 }
```

הפונקציה cout.fill שומרת על התו המוגדר בה, עד לקריאה נוספת של הפונקציה ובה תו מילוי אחר.

הצגת מספר הספרות של משתנה float

לא ניתן לקבוע מראש את מספר הספרות העשרוניות ש- cout יציג עבור ערך float. אולם, על ידי שימוש בפונקציה setprecision ניתן לקבוע את מספר הספרות העשרוניות הרצוי להצגת המספר. בתוכנית **SETPREC.CPP** מתואר השימוש בפונקציה setprecision לקביעת מספר הספרות שיוצגו מימין לנקודה העשרונית.

```
#include <iostream.h>
#include <iomanip.h>

void main(void)
 {
   float value = 1.23456;

   int i;

   for (i = 1; i < 6; i++)
     cout << setprecision(i) << value << endl;
 }
```

לאחר הידור והרצת התוכנית נראה על המסך את השורות הבאות:

```
C:\> SETPREC  <Enter>
1.2
1.23
1.235
1.2346
1.23456
```

הפונקציה setprecision שומרת את מספר הספרות העשרוניות להצגה, ומשתמשת בערך זה לקריאה נוספת של הפונקציה שבה יש שינוי ההגדרה.

קלט ופלט של תווים בודדים

לפעמים צריך להציג תווים זה אחר זה, או לקלוט תווים בודדים מהמקלדת. כדי להציג תו אחד בכל פעם, ניתן להשתמש בפונקציה cout.put. בתוכנית **COUTPUT.CPP** מתואר השימוש בפונקציה זו, אשר מציגה על המסך את המשפט: !Rescued by C++ - תו אחר תו.

```
#include <iostream.h>

void main(void)
 {
   char string[] = "Rescued by C++!";
   int i;

   for (i = 0; string[i]; i++)
     cout.put(string[i]);
 }
```

בספריית הפונקציות הסטנדרטיות קיימת הפונקציה toupper, המחזירה את האות הגדולה המקבילה לאות שמתקבלת בה כפרמטר. בתוכנית **COUTUPPR.CPP** מוצג השימוש בפונקציה זו, כדי להמיר את האותיות המתקבלות ואחר כך להציג אותן על המסך על ידי הפונקציה cout.put.

```
#include <iostream.h>
#include <ctype.h>    // toupper prototype

void main(void)
 {
   char string[] = "Rescued by C++!";
   int i;

   for (i = 0; string[i]; i++)
     cout.put(toupper(string[i]));

   cout << endl << "Ending string: " << string << endl;
 }
```

לאחר הידור והרצת התוכנית נראה על המסך את השורות הבאות:

```
C:\> COUTUPPR   <Enter>
RESCUED BY C++!
Ending string: Rescued by C++!
```

קליטה של תווים בודדים מהמקלדת

כשם שהפונקציה cout.put מציגה על המסך תו אחר תו, כך גם הפונקציה cin.get קולטת תו אחר תו. הפונקציה cin.get קולטת את התו מהמקלדת ובמשפט ההשמה אנו מעבירים את הערך הזה למשתנה.

```
letter = cin.get();
```

במהלך ביצוע התוכנית **CIN_GET.CPP** מוצגת על המסך הודעה המבקשת לתת תשובה מהסוג Y או N (כן/לא). בתוכנית מתבצעת לולאה לקבלת התשובה. הלולאה מסתיימת כאשר מתקבלת אחת האותיות Y או N.

```
#include <iostream.h>
#include <ctype.h>

void main(void)
 {
   char letter;

   cout << "Do you want to continue? (Y/N): ";
```

```
   do {
     letter = cin.get();
     // Convert to uppercase
     letter = toupper(letter);
   } while ((letter != 'Y') && (letter != 'N'));

   cout << endl << "You selected " <<letter<< endl;
 }
```

קליטת רצף תווים מהמקלדת

כאשר משתמשים ב- cin לקליטת נתונים ברצף, דרוש תו הפרדה (רווח, TAB או Enter) בין הערכים הנקלטים. במקרים רבים צריך לקלוט בתוכניות רצף של תווים (שורה - line) ולהציב אותם במשתנה מטיפוס מחרוזת. לצורך זה ניתן להשתמש בפונקציה cin.getline. הפרמטרים של הפונקציה הם משתנה המחרוזת המיועד לאחסון התווים וגודל המחרוזת. הנה דוגמה:

```
cin.getline(string, 64);
```

מכיון שהפונקציה cin.getline קולטת את התווים מהמקלדת, יש להקפיד שמספר התווים הנקלטים לא יחרוג מגודל המחרוזת. הדרך המומלצת לציין את גודל המחרוזת היא השימוש באופרטור sizeof, כפי שנראה בדוגמה הבאה:

```
cin.getline(string, sizeof(string));
```

כאשר רוצים לשנות את גודל המחרוזת במועד מאוחר יותר, אין צורך לשנות כל משפט cin.get בתוכנית. נראה שהאופרטור sizeof משתמש בגודל הנכון של המחרוזת בזמן הביצוע. בתוכנית **GETLINE.CPP** מתואר השימוש בפונקציה cin.getline לקליטת רצף תווים מהמקלדת.

```
#include <iostream.h>

void main(void)
 {
   char string[128];

   cout << "Type line of text and press Enter" << endl;

   cin.getline(string, sizeof(string));

   cout << "You typed: " << string << endl;
 }
```

בביצוע קליטה של מחרוזת מהמקלדת, יש צורך לפעמים לקלוט תווים עד למילוי המחרוזת, או עד לקליטת תו מסוים. כדי לבצע משימה זו, ניתן להעביר פרמטר נוסף לפונקציה cin.getline הכולל בתוכו את התו המסוים הזה. לדוגמה, המשפט הבא קורא לפונקציה cin.getline לקליטת שורה. הקליטה מסתיימת כאשר המשתמש לוחץ על המקש Enter, כאשר נקלטים 64 תווים, או כאשר נקלטת האות Z.

```
cin.getline(string, 64, 'Z');
```

בתוכנית הבאה **UNTIL_Z.CPP** מתואר שימוש בפונקציה cin.getline כאשר מתבצעת קליטה של שורת תווים. על פי פרמטר הפונקציה, הקליטה מסתיימת כאשר לוחצים על האות Z במקלדת.

מומלץ להדר את התוכנית ולהריץ אותה מספר פעמים. בפעם הראשונה לקלוט שורה המכילה את האות Z בסוף השורה; פעם שנייה - כאשר האות Z מופיעה ישר בהתחלת השורה. פעם נוספת כאשר אין האות Z מופיעה כלל בשורת הקלט.

```
#include <iostream.h>

void main(void)
 {
   char string[128];

   cout << "Type line of text and press Enter" << endl;

   cin.getline(string, sizeof(string), 'Z');

   cout << "You typed: " << string << endl;
 }
```

סיכום

בכל תוכנית הכתובה בשפת ++C קרוב לוודאי שיהיה צורך להשתמש בעצמים cin ו-cout לביצוע פעולות קלט/פלט. בפרק זה ראינו מספר פונקציות לטיפול בנתונים, ופונקציות הפועלות בערוצים cin ו- cout. ככל שהתוכניות נעשות מורכבות יותר, הן תשמורנה נתונים בקבצים. בפרק 34 נלמד על פעולות קלט ופלט בקבצים.

לפני שנמשיך, נבחן אם מובנים הנושאים הבאים:

- ✓ cin ו-cout הם עצמים של המחלקות istream ו-ostream המוגדרות בקובץ הכותר iostream.h. הם מציעים מיגוון של פונקציות המאפשרות לבצע פעולות קלט/פלט שונות בתוכנית.
- ✓ הפונקציה cout.width מאפשרת לקבוע מספר תווים (פוזיציות) עבור ערך הפלט הנרשם לאחר הפונקציה.

- ✓ הפונקציה cout.fill מאפשרת לציין תו כלשהו שיוצב כתו מילוי על ידי הפונקציה cout.width, או המשפט setw, במקום תווי רווח.
- ✓ הפונקציה setprecision מאפשרת לקבוע את מספר הספרות העשרוניות להצגת ערך מטיפוס float.
- ✓ הפונקציות cin.get ו- cout.put מאפשרות קליטה והצגה של תווים בודדים.
- ✓ הפונקציה cin.getline מאפשרת לקלוט שורת תווים (רצף) מהמקלדת. עם אפשרות להגבלת מספר תווים, או עד לקליטת תו כלשהו.

תרגילים

1. כיתבו תוכנית המקבלת מהמשתמש כקלט שני מספרים שלמים nb_lines ו- line_width ומדפיסה nb_lines שורות של תווי '$' באורך של line_width תווים כל אחת.
2. כיתבו תוכנית המדפיסה את Π, ולמעשה את הקירוב על פי 7/22, בדיוק של i=1..10 ספרות אחרי הנקודה. כל התוצאות צריכות להיות מיושרות מימין, כך שהספרה האחרונה היא בעמודה ה-20 במסך.
3. כיתבו תוכנית הקולטת 10 תווים. לאחר מכן מדפיסה כל תו בשורה נפרדת. התו במקום i מודפס בעמודה i, ואם הוא גדול מהתו הגדול ביותר עד כה, הוא אינו מודפס אלא מודפסת כוכבית במקומו.
4. כיתבו תוכנית אשר קוראת n תווים (באנגלית) מהקלט, עד אשר:
   א. הוזנו כבר **n** תווים
   ב. המשתמש הזין את התו נקודה ('.').

התוכנית תציג כעת את התווים שהוזנו. כל אות הקטנה מ-'i', היא תציג באותיות (אנגלית) גדולות.

פרק 34

פעולות קלט/פלט בקבצים

תוכניות מורכבות כוללות פעולות של אחסון ואחזור מידע מקבצים. בנושא טיפול בקבצים, כמו בהרבה תחומים אחרים בשפה ניתן להשתמש בתוכניות הכתובות ב-C++, בטכניקות המוכרות משפת C. המהדרים של C++ מספקים קבוצות של מחלקות לביצוע פעולות קלט/פלט בקבצים, ומקלים בכך על ביצוע פעולות קלט פלט בקבצים.

בהמשך נדון בנושאים הבאים:

- פתיחת ערוץ פלט לקובץ ואחסון מידע על ידי שימוש באופרטור הפלט '<<' (Insertion operator).
- פתיחת ערוץ קלט לקובץ וקריאת המידע על ידי שימוש באופרטור הקלט '>>' (Extraction operator).
- פתיחה וסגירה של קבצים על ידי שימוש בפונקציות המחלקה.
- קריאה וכתיבה של נתונים על ידי שימוש בפונקציות המחלקה.

רוב התוכניות, ובמיוחד אלו המעבדות נתונים לשימוש חוזר, מאחסנות את הפלט בקבצים כדי לקרוא אותו בעת העיבוד הבא. לכן, מומלץ להשקיע זמן ולהעמיק בנושא המוצג בפרק זה. כפי שנראה בהמשך - פעולות קלט/פלט בקבצים פשוטות למדי.

ערוץ פלט לקובץ

בפרק 33 למדנו ש-cout הוא עצם מטיפוס ostream (ערוץ פלט). על ידי שימוש במחלקה ostream התוכנית מסוגלת לבצע פעולות בערוץ הפלט, כשהיא משתמשת באופרטור הפלט ובאופרטורים והפונקציות של המחלקה. בצורה דומה, קובץ הכותר fstream.h מגדיר מחלקה בשם ofstream. השימוש בעצמים של המחלקה ofstream מאפשר לבצע פעולות פלט לקובץ. כדי להשתמש במחלקה זו, צריך ליצור תחילה עצם מטיפוס ofstream ולציין (בין גרשיים) את שם קובץ הפלט, לפי הדוגמה הבאה:

```
ofstream  file_object("FILENAME.EXT");
```

שם הקובץ הנמצא במשפט יצירת העצם מטיפוס ofstream גורם ל**דריסת** (**Overwrite**) קובץ בשם זה הקיים בדיסק, או ליצירת קובץ חדש, אם קובץ זה לא נמצא על הדיסק. בתוכנית **OUT_FILE.CPP** מתואר השימוש בעצם מטיפוס ofstream ליצירת קובץ מסוים, אשר בהמשך התוכנית משתמשת באופרטור הפלט לכתיבת מספר שורות טקסט בקובץ.

```
#include <fstream.h>

void main(void)
 {
   ofstream book_file("BOOKINFO.DAT");

   book_file << "Rescued by C++" << endl;
   book_file << "Jamsa Press" << endl;
   book_file << "315 pages" << endl;
 }
```

התוכנית הקודמת פתחה קובץ בשם **BOOKINFO.DAT**, ואחר כך היא כתבה בו שלוש שורות טקסט. לאחר הידור והרצת התוכנית ניתן להשתמש בפקודה TYPE של מערכת הפעלה DOS כדי להתבונן בתוכן הקובץ שנוצר.

```
C:\> TYPE  BOOKINFO.DAT   <Enter>
Rescued by C++
Jamsa Press
315 pages
```

ערוץ קלט לקובץ

בסעיף הקודם הצגנו את השימוש בעצם מטיפוס ofstream אשר מקל על ביצוע פעולות בערוץ הפלט לקבצים. באופן דומה ניתן לבצע פעולות בערוץ הקלט לקבצים על ידי שימוש בעצם מטיפוס ifstream. כדי להשתמש במחלקה זו, יש ליצור תחילה עצם מטיפוס ifstream ולציין את שם הקובץ המשמש לקלט, לפי הדוגמה שלהלן:

```
ifstream input_file("FILENAME.EXT");
```

בתוכנית **FILE_IN.CPP** פותחים את הקובץ BOOKINFO.DAT שנוצר בתוכנית הקודמת. בהמשך קוראים ומציגים את שלושת האלמנטים הראשונים המאוחסנים בקובץ.

```
#include <iostream.h>
#include <fstream.h>

void main(void)
 {
   ifstream input_file("BOOKINFO.DAT");

   char one[64], two[64], three[64];

   input_file >> one;
   input_file >> two;
   input_file >> three;

   cout << one << endl;
   cout << two << endl;
   cout << three << endl;
 }
```

לאחר הידור והרצת התוכנית אנו מצפים להצגת שלוש השורות הראשונות של הקובץ. אולם כמו בפקודת הקריאה הרגילה cin, גם ערוץ הקלט משתמש בתווי הפרדה בין האלמנטים הנקלטים, ולכן הרצת התוכנית תראה על המסך את השורות הבאות:

```
C:\> FILE_IN  <Enter>
Rescued
by
C++!
```

קיבלנו שלוש מילים בשורות נפרדות ולא שלוש שורות בנות מספר מילים כל אחת. המסקנה - צריך להשתמש בפקודה אחרת לקריאת השורה.

קריאה של שורה מתוך קובץ

בפרק 33 הצגנו את פונקציית המחלקה cin.getline, שמאפשרת לקלוט שורה שלמה מהמקלדת. בצורה דומה ניתן להפעיל את הפונקציה getline של המחלקה ifstream, כדי לקרוא שורה שלמה מתוך קובץ. בתוכנית **FILELINE.CPP** מוצג השימוש בפונקציה getline לקריאת שלוש השורות המאוחסנות בקובץ **bookinfo.dat**.

```
#include <iostream.h>
#include <fstream.h>

void main(void)
 {
   ifstream input_file("BOOKINFO.DAT");

   char one[64], two[64], three[64];

   input_file.getline(one, sizeof(one));
   input_file.getline(two, sizeof(two));
   input_file.getline(three, sizeof(three));

   cout << one << endl;
   cout << two << endl;
   cout << three << endl;
 }
```

תוכנית זו מצליחה לקרוא את תוכן הקובץ במלואו, מכיון שהיה ידוע לה שבקובץ מאוחסנות שלוש שורות. ברוב המקרים אין המתכנת יודע את מספר השורות הקיימות בקובץ שהוא קורא ממנו. לכן, צריך לקרוא את שורות הקובץ עד למציאת ציון **סוף קובץ (End of file)**.

בדיקת ציון סוף הקובץ

בדיקת ציון **סוף קובץ (EOF - End Of File)** היא אחת הפעולות הנפוצות ביותר כאשר קוראים תוכן של קובץ. כדי לערוך בדיקה זו ניתן להשתמש בפונקציית המחלקה eof. הפונקציה מחזירה את הערך אפס כל עוד לא נמצא ציון סוף הקובץ, ולחילופין היא מחזירה את הערך 1, כאשר התנאי מתקיים ונמצא ציון סוף הקובץ. ניתן להיעזר בלולאת while כדי לבצע פעולות קריאה חוזרות של תוכן הקובץ עד למציאת ציון סוף קובץ.

```
while (! input_file.eof())
 {
   // Statements
 }
```

הלולאה הקודמת מתבצעת עד שהפונקציה eof תחזיר את הערך אפס. בתוכנית **TEST_EOF.CPP** משתמשים בפונקציה eof כדי לקרוא את תוכן הקובץ **bookinfo.dat** עד למציאת ציון סוף הקובץ.

```
#include <iostream.h>
#include <fstream.h>

void main(void)
 {
   ifstream input_file("BOOKINFO.DAT");

   char line[64];

   while (! input_file.eof())
    {
      input_file.getline(line, sizeof(line));

      cout << line << endl;
    }
 }
```

בצורה דומה, בתוכנית **WORD_EOF.CPP** קוראים את המילים הכלולות בקובץ, עד למציאת ציון סוף הקובץ.

```
#include <iostream.h>
#include <fstream.h>

void main(void)
 {
   ifstream input_file("BOOKINFO.DAT");

   char word[64];

   input_file >> word;
   while (! input_file.eof())
    {
      cout << word << endl;  // ENDL
      input_file >> word;

      cout << word << endl;
    }
 }
```

בתוכנית **CHAR_EOF.CPP** לדוגמה, קוראים את התווים הכלולים בקובץ עד למציאת ציון סוף הקובץ. ראה להלן.

```
#include <iostream.h>
#include <fstream.h>

void main(void)
 {
   ifstream input_file("BOOKINFO.DAT");

   char letter;

   letter = input_file.get();
   while (! input_file.eof())
    {
      cout << letter;
      letter = input_file.get();
    }
 }
```

בדיקת שגיאה בפעולות קלט/פלט בקבצים

בתוכניות קודמות איננו מניחים התרחשות תקלה כלשהי במהלך ביצוע פעולות קלט/פלט. לצערנו אין זה נכון לנהוג כך. למשל, כשפותחים קובץ, יש לוודא כי הקובץ אכן קיים בדיסק. כאשר כותבים לקובץ, יש לוודא שהפעולה הסתיימה בהצלחה, עלול להיות מצב שהדיסק מלא ואי אפשר לכתוב לקובץ. ניתן להשתמש בפונקציית המחלקה fail כדי לבדוק תקלות בביצוע פעולות קלט/פלט בקבצים. הפונקציה מחזירה את הערך **0** (**אפס**) כאשר מתרחשת תקלה בביצוע פעולה כלשהי. לעומת זאת, במקרה שלא מתרחשת תקלה הערך המוחזר יהיה **1** (**אחד**). לדוגמה, ניתן להשתמש בקטע תוכנית הבא הנעזר בפונקציה fail כדי להתריע על תקלה בעת פתיחת הקובץ:

```
ifstream input_file("FILENAME.DAT");

if (input_file.fail())
 {
    cerr << "Error opening FILENAME.EXT" << endl;
    exit(1);
 }
```

בדרך דומה ניתן לכתוב קטע תוכנית שבודק אם פעולות הקריאה והכתיבה לקובץ מתבצעות בהצלחה. בתוכנית **TEST_ALL.CPP** מתואר השימוש בפונקציית המחלקה fail לאיתור תקלות שונות - ראה להלן.

```
#include <iostream.h>
#include <fstream.h>

void main(void)
 {
   char line[256];

   ifstream input_file("BOOKINFO.DAT");

   if (input_file.fail())
    cerr << "Error opening BOOKINFO.DAT" << endl;
   else
    {
     while((! input_file.eof()) &&  (! input_file.fail()))
         {
           input_file.getline(line, sizeof(line));

           if (! input_file.fail())
             cout << line << endl;
         }
    }
 }
```

סגירת קובץ

בסיום הרצת התוכנית, מערכת ההפעלה סוגרת את כל הקבצים שנשארו פתוחים (אם לא נסגרו קודם לכן על ידי התוכנית). למרות זאת, טוב לסגור קבצים אשר אין להן שימוש במהלך ביצוע התוכנית. כדי לסגור קבצים יש להשתמש בפונקציה close כפי שמוצג כאן:

```
input_file.close();
```

בעת סגירת הקובץ מתבצע עדכון של פרטי הקובץ בטבלת ניהול הקבצים של מערכת ההפעלה.

בחירת צורת הפתיחה של הקובץ

בתוכניות הקודמות התבצעו פעולות הקלט/פלט בקבצים חדשים. במקרים רבים נרצה לכתוב את הנתונים בקובץ פלט בהמשך לנתונים הקיימים בו ועל כן יש לפתוח את הקובץ באופן שהמידע יתוסף בסופו. כדי לבצע זאת, ניתן לקבוע את צורת הפתיחה של הקובץ על ידי ציון קוד מתאים בפרמטר השני במשפט יצירת העצם. הנה דוגמה:

```
ifstream output_file("FILENAME.EXT", ios::app);
```

הפרמטר ios::app מציין את אופן הפתיחה של הקובץ. בטבלה 34.1 מוצגים הערכים המקובלים לציון אופן הפתיחה של קבצים.

טבלה 34.1: ערכי קוד הפתיחה הקבצים.

קוד הפתיחה	מטרה
ios::app	פתיחת הקובץ להוספה והצבת המצביע של הקובץ בסוף הקובץ.
ios::ate	הצבת המצביע של הקובץ בסוף הקובץ.
ios::in	פתיחת הקובץ לקריאה.
ios::nocreate	אם הקובץ אינו קיים, הוא אינו נוצר והפונקציה מחזירה ערך המציין תקלה.
ios::noreplace	אם הקובץ קיים, פעולת הפתיחה נכשלת, והפונקציה מחזירה ערך המציין תקלה.
ios::out	פתיחת הקובץ לכתיבה.
ios::trunc	דריסה של (כתיבה על...) תוכן הקובץ הקיים.

הערה


הכתיבה לקובץ או הקריאה ממנו, מבוקרת באמצעות **מצביע (pointer)** של הקובץ אל המקום שבו צריך לכתוב או לקרוא ממנו נתונים. בעת פתיחת קובץ המצביע נמצא בתחילתו. כדי להוסיף נתונים בסוף קובץ קיים, או לקרוא נתונים מקובץ קיים, עלינו להעביר את המצביע אל המקום הרצוי בקובץ. עושים זאת בהוראה מתאימה על ידי הפרמטר השני ios:ate או ios:app, כפי שנראה בהמשך.

במשפט הבא פותחים את הקובץ לכתיבה. קוד הפתיחה ios:noreplace מונע כתיבה על קובץ קיים בשם זה (זוהי הגנה מפני טעות בכתיבת שם הקובץ).

```
ifstream output_file("FILENAME.EXT", ios::out | ios::noreplace);
```

פעולות קריאה וכתיבה

בתוכניות שהוצגו עד כאן התייחסו פעולות הקריאה וכתיבה בקובץ למחרוזות תווים. בתוכניות מורכבות יותר פעולות קריאה וכתיבה בקובץ תתייחסנה למערכים או רשומות. לפעולות בסוג זה של נתונים יש להפעיל את הפונקציות read ו-write. לפני שנוכל לעשות זאת, עלינו להגדיר מאגר בזיכרון אשר יטפל בנתונים, וגם לציין את גודלו הנמדד בבתים.

```
input_file.read(buffer, sizeof(buffer));

output_file.write(buffer, sizeof(buffer));
```

בתוכנית **STRU_OUT.CPP** מתואר השימוש בפונקציה write כדי לכתוב בקובץ **employee.dat** את תוכן מבנה הרשומה employee.

```
#include <iostream.h>
#include <fstream.h>

void main(void)
 {
  struct employee {
    char name[64];
    int age;
    float salary;
   } worker = { "John Doe", 33, 25000.0 };

  ofstream emp_file("EMPLOYEE.DAT" );

  emp_file.write((char *) &worker, sizeof(employee));
 }
```

הפונקציה write מקבלת מצביע למחרוזת תווים, ולכן משתמשים בביטוי המתאם (* char) כדי לאפשר העברה לפונקציה של מצביע מטיפוס אחר. בצורה דומה, בתוכנית **STRU_IN.CPP** מתואר השימוש בפונקציה read לקריאת רשומת עובד מתוך הקובץ.

```
#include <iostream.h>
#include <fstream.h>

void main(void)
 {
  struct employee {
    char name[64];
    int age;
    float salary;
   } worker;

  ifstream emp_file("EMPLOYEE.DAT");

  emp_file.read((char *) &worker, sizeof(employee));

  cout << worker.name << endl;
  cout << worker.age << endl;
  cout << worker.salary << endl;
 }
```

סיכום

בתוכניות שהן יותר מאשר תרגילי תכנות, דרושות פעולות בקבצים, המהווים את מאגר המידע של התוכנית.

לפני שנעבור לפרק הבא, נבחן אם מובנים לנו הנושאים הבאים:

- ✓ בקובץ כותר fstream.h מוגדרות המחלקות ifstream ו-ofstream, המאפשרות לבצע פעולות קלט/פלט בקבצים.
- ✓ כדי לפתוח קובץ לפעולות קלט/פלט יש ליצור עצם מטיפוס ifstream או מטיפוס ofstream, ולהעביר לבנאי המחלקה את שם הקובץ הרצוי.
- ✓ לאחר פתיחת הקובץ לקריאה או כתיבה, ניתן להשתמש באופרטורים '<<' (extraction) או '>>' (insertion) כדי לקרוא נתונים מהקובץ או לכתוב בו נתונים, בהתאמה.
- ✓ הפונקציות get ו-put מאפשרות לקרוא תו בודד מקובץ ולכתוב תו בודד לקובץ, בהתאמה.
- ✓ הפונקציה getline מאפשרת לקרוא שורה שלמה מתוך הקובץ.
- ✓ הפונקציה eof מאפשרת לבדוק את ציון סוף הקובץ. בדיקה זו חשובה מאוד, מכיון שבדרך כלל מתבצעת בתוכנית סריקה של קבצים עד למציאת ציון סוף הקובץ.
- ✓ הפונקציה fail מאפשרת לבדוק את הסיום המוצלח של פעולות קלט/פלט בקבצים.
- ✓ הפונקציות read ו-write מיועדות לקריאה, כתיבה וטיפול בקבצים המאחסנים מערכים או רשומות.
- ✓ בסוף הטיפול בקובץ מומלץ לסגור אותו על ידי שימוש בפונקציה close.

בפרק 35 נציג כיצד לשפר את ביצועי התוכניות על ידי שימוש ב**פונקציות משולבות** (**Inline functions**).

תרגילים

1. מה עושה התוכנית הבאה:

```
#include <iostream.h>
#include <fstream.h>

void main(void)
  {
      char c;
      ifstream f(__FILE__);
      while ((!f.fail()) && (!f.eof()))
      {
            f.read(&c, 1);
            cout << c;
      }
  }
```

2. כיתבו תוכנית הכוללת מחלקה בשם people, בעלת הנתונים הבאים: שם פרטי, שם משפחה, וגיל. כיתבו פונקציות לכתיבה וקריאה של תוכן האובייקט אל קבצים. הגדירו פונקציות read ו-write מתאימות.

3. כיתבו את הפעולות של התרגיל הקודם על ידי העמסת האופרטורים >>, << בפעולה עם קובץ:

```
friend ostream& operator<<(ostream& o, people& p);
friend ostream& operator>>(istream& o, people& p);
```

4. כיתבו תוכנית המאפשרת למשתמש לכתוב הערות לקובץ. התוכנית תבקש מהמשתמש שם קובץ לעריכה, ואז תציג את הקובץ אם הוא קיים, ותאפשר למשתמש להקליד הערות לסופו של הקובץ, עד להקשת צירוף התווים control-D (char 04).

פרק 35

פונקציות משולבות וקוד אסמבלי בתוכנית

בתוכניות ובמחלקות שהוצגו מהפרק השמיני ועד כאן השתמשנו בצורה מורחבת בפונקציות. למדנו שהחיסרון הבולט של שימוש בפונקציות הוא **תקורת זמן הביצוע** (**Overhead**), כלומר – הארכת משך זמן ההרצה הנובעת מהעתקת הפרמטרים למחסנית בכל פעם שקוראים לפונקציה. בפרק זה נציג שיטה מיוחדת להגדרת פונקציה המונעת את תהליך העתקת הפרמטרים של הפונקציה למחסנית שיטת **הקוד המשולב** (Inline code). דבר זה מוריד בצורה משמעותית את זמני הביצוע והתגובה של התוכנית על ידי חיסכון של זמן התקורה של הקריאה לפונקציה. פונקציות אלו נקראות **פונקציות משולבות** (**Inline functions**).

בהמשך הפרק נדון בנושאים הבאים:

- כדי לשפר ביצועים על ידי חיסכון בזמן התקורה לקריאה לפונקציות, אפשר להורות למהדר ++C לשלב את קוד הפונקציה בין פקודות התוכנית, בדומה לפיתוח פקודות המאקרו במקום שהמאקרו נכתב.
- באמצעות פונקציות משולבות, התוכנית מובנת יותר וגם ארוכה יותר (הפונקציות מפורטות במקום שהן דרושות), אך נמנעת התקורה שנובעת מהעברה ושליפה של פרמטרים למחסנית כאשר מבצעים קריאה לפונקציה.
- כתלות בדרישות התוכנית, צריך לפעמים לצרף שגרות בשפת אסמבלי כדי להשלים מטלות תכנות מסוימות.
- כדי לפשט את השימוש בתוכניות בשפת אסמבלי, אפשר להגדיר בתוכנית ++C פונקציות אסמבלי משולבות.

כפי שלמדנו, הדרך הטובה לשלוט בשפת ++C היא לבצע ניסיונות שונים וחוזרים בהרצת התוכניות המוצגות בספר. לכן נחזור ונמליץ להשקיע זמן בתרגול הנושאים שבפרק זה.

פונקציות משולבות

כאשר מגדירים פונקציה בתוך התוכנית, מהדר שפת ++C ממיר את קוד הפונקציה לפקודות בשפת מכונה, והוא שומר העתק אחד בלבד של פקודות אלו בתוכנית. בכל פעם שהתוכנית קוראת לפונקציה, המהדר מציב במקום הקריאה סדרה של פקודות המיועדות להעברת הפרמטרים של הפונקציה למחסנית לצורך הביצוע. בסיום ביצוע הפונקציה, התוכנית ממשיכה במשפט הראשון הסמוך למשפט הקריאה לפונקציה. העברת הפרמטרים למחסנית וקפיצת התוכנית למקום בו נמצא קוד הביצוע של הפונקציה מוסיף לתוכנית **תקורת זמן** (**Overhead**). דבר זה גורם להאטת ביצוע התוכנית. לחילופין נניח שבכל מקום שבו קוראים לפונקציה, המהדר היה מציב עותק של פקודות הפונקציה ומשתמש בפרמטרים הזמינים לו מבלי לטרוח להעביר אותם למחסנית. כלומר, אנו **משלבים** את קוד הפונקציה בקוד התוכנית. כעת נתבונן בתוכנית **CALLBEEP.CPP**. בתוכנית זו מתבצעות שתי קריאות לפונקציה show_message המציגה הודעה על המסך תוך השמעת מספר צפצופים ברמקול המחשב.

```
#include <iostream.h>

void show_message(int count, char *message)
 {
   int i;

   for (i = 0; i < count; i++)
     cout << '\a';

   cout << message << endl;
 }

void main(void)
 {
   show_message(3, "Rescued by C++");

   show_message(2, "Lesson 35");
 }
```

בתוכנית הבאה **NO_CALL.CPP**, אין קריאות לפונקציה show_message. במקום בו היו צריכות להתבצע קריאות לפונקציה, מוצבים המשפטים של הפונקציה עצמה. בדוגמה זו מופיע הקוד של הפונקציה פעמיים - ראה להלן.

```
#include <iostream.h>

void main(void)
 {
   int i;

   for (i = 0; i < 3; i++)
     cout << '\a';
```

```
    cout << "Rescued by C++" << endl;

  for (i = 0; i < 2; i++)
    cout << '\a';

  cout << "Lesson 35" << endl;
}
```

שתי התוכניות הקודמות פועלות בצורה זהה ומפיקות את אותו הפלט. מכיון שבתוכנית no_call לא מבצעים קריאות לפונקציה show_message, בביצוע מהיר ביחס למהירות הביצוע של תוכנית callbeep. במקרה זה קשה להבחין בהבדלי מהירות הביצוע מכיון שהתוכניות פשוטות ויש בהן שתי קריאות בלבד לפונקציה. אולם בתוכנית שקוראים בה 1000 פעמים לפונקציה, ניתן להבחין בשיפור הביצוע. התוכנית no_call מהירה יותר מתוכנת callbeep ולעומת זאת, התוכנית האחרונה ברורה ומובנת יותר. מן הראוי לציין, שהתוכנית no_call עלולה להיות ארוכה יותר, מכיון שקוד הפונקציה מופיע פעמיים, ולא פעם אחת, כמו בתוכנית callbeep.

בתהליך כתיבת מערכות מחשב עולה השאלה מתי להשתמש בפונקציות רגילות ומתי כדאי יותר להשתמש בפונקציות משולבות. התשובה תלויה בדרישות התוכנית: ניתן לומר, שבתוכניות רגילות שימוש בפונקציות רגילות טוב יותר. לעומת זאת, כאשר זמני התגובה בתוכנית חשובים במיוחד מומלץ להשתמש בפונקציות משולבות, מכיון שהן מורידות את הזמן המבוזבז בתהליכי הקריאה לפונקציה. אחת הדרכים המוצעות להורדת תקורת הקריאה לפונקציות מתוארת בתוכנית **NO_CALL**. כפי שאנו רואים, יישום זה מוריד את הבהירות ואת יכולת ההבנה של התוכנית. אך למזלנו, שפת C++ מכילה את המילה השמורה inline, המאפשרת לנו ליהנות מהטוב שבשני העולמות.

המילה השמורה inline

שפת C++ מאפשרת להציב לפני כל הגדרת פונקציה את המילה השמורה inline. בשלב ההידור, יוצב עותק מלא של משפטי הפונקציה בכל מקום בו מתבצעת קריאה לפונקציה זו. המהדר יכתוב את המשפטים על פי הפרמטרים שבקריאה לפונקציה. בצורה זו אין מורידים מבהירות התוכנית, אך משפרים את זמני הביצוע שלה, כי מורידים את תקורת הזמן (Overhead) הקשורה בתהליכי הקריאה לפונקציה. בתוכנית **INLINE.CPP** מוגדרות שתי הפונקציות min ו-max כפונקציות משולבות.

```
#include <iostream.h>

inline int max(int a, int b)
 {
   if (a > b)
     return(a);
   else
     return(b);
 }
```

```
inline int min(int a, int b)
 {
   if (a < b)
     return(a);
   else
     return(b);
 }

void main(void)
 {
   cout << "The min of 1001 and 2002 is "
                           << min(1001, 2002) << endl;
   cout << "The max of 1001 and 2002 is "
                           << max(1001, 2002) << endl;
 }
```

בתוכנית זו, בכל מקום שבו מתבצעת קריאה לפונקציות המשולבות יוצב עותק מלא של משפטי הפונקציה המתאימה. כך, זמני התגובה של התוכנית משתפרים מבלי לגרוע ביכולת הבנתן של התוכנית. נשים לב לכך שתוכנית המקור נשארת ללא שינוי והיא ברורה, כפי שרצינו. השינוי הוא של המהדר בתוכנית הביצוע.

פונקציה משולבת, מה היא?

בכל מקום בו מתבצעת קריאה לפונקציה זו יוצב עותק מלא של משפטי הפונקציה תוך התאמת משפטים אלה לפרמטרים הנדרשים. הדבר קורה כאשר המהדר מזהה את המילה inline לפני הגדרת הפונקציה. בדרך זו מקבלים פונקציה משולבת (Inline function) בקוד התוכנית בכל מקום שהיא דרושה, לעומת פונקציה שנמצאת בתוכנית פעם אחת בלבד ו"קופצים" אליה לשם ביצוע. בדרך זו משפרים את זמני הביצוע של התוכנית על ידי הורדת תקורת הזמן הקשורה בתהליכי הקריאה לפונקציה. יחד עם זאת, נשמרת בהירות התוכנית (מן הראוי לשים לב שהתוכנית עלולה להיות ארוכה יותר כתוצאה משכפול הקוד של הפונקציה פעמים רבות).

פונקציות משולבות במחלקה

ניתן להגדיר פונקציות מחלקה בתוך המחלקה עצמה, או מחוצה לה. לדוגמה, במחלקה employee פונקציות המחלקה מוגדרות בתוך המחלקה.

```
class employee {
 public:
   employee(char *name, char *position, float salary)
    {
      strcpy(employee::name, name);
      strcpy(employee::position, position);
```

```
      employee::salary = salary;
    }
   void show_employee(void)
    {
      cout << "Name: " << name << endl;
      cout << "Position: " << position << endl;
      cout << "Salary: $" << salary << endl;
    }
 private:
   char name[64];
   char position[64];
   float salary;
};
```

הגדרת פונקציות המחלקה בתוך המחלקה עצמה היא הדרך הרגילה ליצירת פונקציות משולבות בתוך המחלקה. בצורת הגדרה זו, קוד הפונקציה מועתק לכל עצם שנוצר מטיפוס המחלקה. יתרון השיטה הוא שיפור בביצוע התוכנית. החיסרון הוא שהתוכנית גדלה ככל שמרבים ביצירת עצמים. חיסרון נוסף הוא הורדת רמת הבהירות של קוד המחלקה, בגלל הוספת הגדרות הפונקציות בתוך הגדרת המחלקה.

כדי לשפר את רמת הבהירות של הגדרת המחלקות, ניתן להוציא את הגדרת הפונקציה מחוץ למחלקה ולהציב לפני שם הפונקציה את המילה השמורה inline. לדוגמה: הגדרת הפונקציה הבאה גורמת למהדר להפעיל את כללי הפונקציה המשולבת עבור הפונקציה show_employee.

```
inline void employee::show_employee(void)
 {
   cout << "Name: " << name << endl;
   cout << "Position: " << position << endl;
   cout << "Salary: $" << salary << endl;
 }
```

שילוב קוד אסמבלי בתוכנית

בפרק ראשון למדנו שניתן לכתוב תוכניות תוך שימוש במיגוון רחב של שפות תכנות. מהדר השפה מתרגם את משפטי התוכנית לקוד מכונה, שהוא השפה שבאמצעותה המחשב מסוגל לבצע את התוכנית. שפת אסמבלי היא שפת ביניים הנמצאת בין שפת תכנות עילית (כמו C, ++C או פסקל) לבין שפת מכונה. שפת אסמבלי משתמשת בקודים ובסמלים שונים המייצגים את פקודות שפת המכונה. לפעמים, לפי דרישות התוכנית צריך לכתוב פקודות באסמבלי המשתלבות בתוכניות הכתובות בשפת ++C. כדי לבצע משימה זו יש להשתמש במשפט asm של שפת ++C. אנו מציגים את המשפט הזה כדי להשלים את המידע אודות השפה. ברוב התוכניות שנכתוב אין צורך לשלב משפטים בשפת אסמבלי.

בתוכנית **USE_ASM.CPP** מתואר השימוש במשפט asm. משפטי אסמבלי שבדוגמה גורמים להשמעת צפצוף ברמקול המחשב הפועל בסביבת עבודה DOS.

שים ♥ ! **תוכנית זו פועלת בגירסה Borland C++ 3.1 ומעלה בלבד.**

```
#include <iostream.h>

void main(void)
 {
   cout << "About to sound the bell!" << endl;

   asm {
     MOV AH,2
     MOV DL,7
     INT Ox21;
   }

   cout << "Done!" << endl;
 }
```

סיכום

פונקציות משולבות (Inline functions) משפרות את ביצועי התוכנית על ידי הקטנת התקורה של השימוש בפונקציות (אין צורך "לקרוא" להן). בפרק זה למדנו כיצד ומתי להשתמש בפונקציות משולבות בתוכניות שונות. למדנו גם שלפעמים צריך לכתוב שגרות בשפת אסמבלי כדי לבצע משימות מיוחדות.

בפרק 36 נראה כיצד התוכניות יכולות לגשת לארגומנטים של שורת-פקודה, אשר המשתמש מקליד לצורך הרצת תוכנית.

אך לפני שנעבור לפרק הבא, הבה נראה אם מובנים לנו הנושאים שעברנו עד כה:

- ✓ העברת פרמטרים של פונקציה לתוך המחסנית וקפיצה למקום בו נמצא קוד הפונקציה הן פעולות שגורמות להאטת ביצוע התוכנית עקב תקורת הזמן (Overhead) הנובע מהן.
- ✓ המילה השמורה inline גורמת שהמהדר יציב עותק של משפטי הפונקציה בכל מקום בתוכנית שבו מתבצעת קריאה לפונקציה. על כן, תקורת הזמן הדרושה לביצוע קריאת הפונקציה נעלמת וזמני ביצוע התוכנית משתפרים, אך התוכנית נעשית ארוכה יותר.
- ✓ השימוש בפונקציות משולבות (Inline functions) בתוך המחלקה גורם לשכפול קוד הפונקציה לכל עצם המחלקה שנוצר בתוכנית. בדרך כלל, עצמי המחלקה משתתפים בעותק אחד של קוד הפונקציה.
- ✓ המילה השמורה asm מאפשרת לשלב משפטים בשפת אסמבלי בתוך תוכנית הכתובה בשפת C++.

תרגילים

1. כיתבו את הפונקציה swap להחלפה של ערכים שלמים (int), פעם כפונקציה משולבת בקוד (inline) ופעם כפונקציה רגילה. בידקו את הפונקציה שנכתבה.

2. התבוננו בתוכנית הבאה, המדגימה שימוש בקוד אסמבלר המשולב בתוכנית C++:

```
void set_color(int color_index, int red, int green, int blue)
  {
// The VGA contains a color palette for each color 0..255
// where each color has a color value (Red,Green,Blue)
// each in the range 0..63.
// This function sets the RGB color for a given palette
// color index, by directing the color index to port 0x3c8,
// and the three RGB values sequencially to port 0x3c9.

      asm mov ax,color_index
      asm mov dx,0x3c8
      asm out dx,al
      asm inc dx
      asm mov ax,red
      asm out dx,al
      asm mov ax,green
      asm out dx,al
      asm mov ax,blue
      asm out dx,al
  }
void main(void)
  {
      int i;
      // color index 0 is the screen-background color
      while (!kbhit()) {
            for (i=0; i<128; i++) {
                  set_color(0, (i>=64) ? 127-i : i ,0
                        ,(i>=64) ? 127-i : i);
                  delay(20);
                  }
            }
    set_color(0, 0, 0, 0);
  }
```

פרק 36

ארגומנטים של שורת הפקודה

כידוע, כאשר מבצעים פקודות דרך שורת הפקודה (System prompt), רוב הפקודות מאפשרות לכלול מידע נוסף בשורת הפקודה, כמו למשל, שם הקובץ עליו הן תפעלנה. לדוגמה, כאשר משתמשים בפקודה COPY של MS-DOS, מציינים בשורת הפקודה גם את שמות שני הקבצים (המקור והמטרה). בדומה, כאשר מפעילים את המהדר דרך שורת הפקודה, יש להוסיף את שם קובץ המקור המיועד להידור. בפרק זה נראה כיצד יכולות תוכניות ++C לקבל את הארגומנטים של שורת הפקודה ולהשתמש בהם. בהמשך הפרק נדון במושגי המפתח הבאים:

- תוכניות ++C מתייחסות לארגומנטים של שורת הפקודה כפרמטרים לפונקציה main.
- ++C מעבירה שניים, ולפעמים שלושה פרמטרים אל הפונקציה main. רוב התוכניות מכנות אותם בשמות argc ו-argv.
- הפרמטר argc מציין את מספר הארגומנטים בשורת הפקודה.
- הפרמטר argv הוא מערך של מחרוזות תווים, המכילות בהתאמה את הפרמטרים של שורת הפקודה.
- על פי המהדר שברשותך, התוכנית יכולה לפנות לפרמטר נוסף, env. פרמטר זה הוא מערך של מחרוזות המכילות את משתני הסביבה.

היכולת לגשת לארגומנטי שורת הפקודה מגדילה את מספר הדרכים בהם ניתן לקרוא לתוכנית כלשהי. למשל, אנו יכולים ליצור תוכנת העתקה (Copy) משלנו, בה נוכל להשתמש להעתקת קובץ מקור כלשהו (המוגדר כארגומנט הראשון בשורת הפקודה) אל קובץ מטרה כלשהו אחר (המוגדר כארגומנט השני).

הגישה ל- argc ו- argv

כאשר מריצים תוכנית דרך שורת הפקודה ב-DOS, שורת הפקודה המוקלדת על ידי המשתמש מועברת לתוכנית:

```
C:\>COPY SOURCE.DOC  TARGET.DOC <Enter>
```

שורת הפקודה מציינת במקרה זה את הפקודה (COPY) וכוללת גם שני ארגומנטים (שם קובץ המקור, SOURCE.DOC, ושם קובץ המטרה, TARGET.DOC). כדי לאפשר לתוכנית לגשת לשורת הפקודה, C++ מעבירה שני פרמטרים לפונקציה main:

```
void  main(int  argc, char *argv[])
```

הפרמטר הראשון, argc, מונה את מספר הכניסות במערך argv. לדוגמה, עבור הפקודה COPY הקודמת, יכיל הפרמטר argc את הערך 3 (שם הפקודה ושני הארגומנטים).

התוכנית הבאה, **SHOWARGC.CPP**, משתמשת בפרמטר argc כדי להציג את מספר הארגומנטים המופיעים בשורת הפקודה:

```
#include <iostream.h>

void main(int argc, char *argv[])
 {
  cout << "Number of command-line arguments is " << argc <<
endl;
 }
```

הנה לפניך דוגמה נוספת להרצת תוכנית, שמזינים לה מספר פרמטרים:

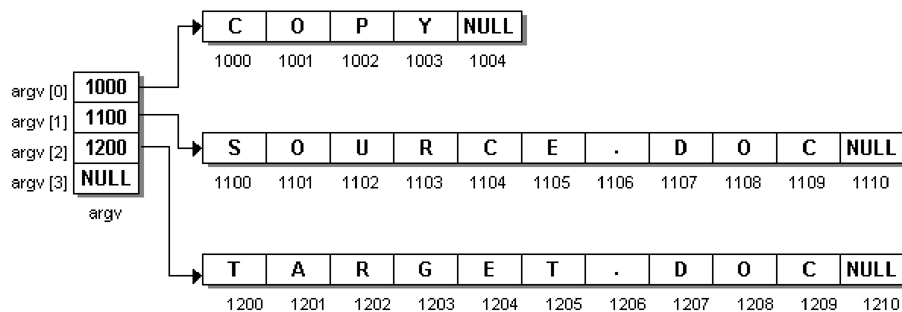
```
C:\>SHOWARGC  A  B  A   <Enter>
```

מומלץ לנסות ולהריץ את התוכנית עם מספר שונה של פרמטרים ולראות מהן התוצאות המתקבלות. המהדר שברשותך עשוי להתייחס אל ארגומנטים המקובצים בין מרכאות כפולות כאל ארגומנט יחיד:

```
C:\>SHOWARGC "This is only one argument"   <Enter>
Number of command-line agruments is 2
```

argv, הפרמטר השני של הפונקציה main, הוא מערך של מחרוזות תווים, המכילות בהתאמה את מרכיבי שורת הפקודה.

תרשים 36.1 ממחיש כיצד כניסות המערך argv מצביעות אל מרכיבים אלה.



תרשים 36.1: המערך argv מצביע אל ארגומנטים של שורת-פקודה.

התוכנית הבאה, **SHOWARGV.CPP**, משתמשת במשפט **for** כדי להציג את כניסות המערך argv (כלומר, את שורת הפקודה של התוכנית). התוכנית מתחילה באיבר הראשון של המערך (שם התוכנית) ואז מציגה את שאר האיברים, כל עוד משתנה הלולאה i קטן מ-argc:

```
#include <iostream.h>

void main(int argc, char *argv[])
 {
  int i;

  for (i = 0; i < argc; i++)
    cout << "argv[" << i << "] contains " << argv[i] << endl;
 }
```

הדרו והריצו את התוכנית הקודמת, והפעילו שורות פקודות שונות. למשל, עבור שורת הפקודה הבאה, נקבל:

```
C:\> SHOWARGV A B C <Enter>
argv[0] contains SHOWARGV.EXE
argv[1] contains A
argv[2] contains B
argv[3] contains C
```

הגישה לארגומנטים שבשורת הפקודה

כדי להגדיל את מיגוון המטלות שתוכניות מסוגלות לבצע, ++C מאפשרת להן לגשת לארגומנטים של שורת הפקודה באמצעות שני פרמטרים המועברים אוטומטית וישירות לפונקציה main.

הפרמטר הראשון, argc, מכיל את מספר הארגומנטים בשורת הפקודה (שם התוכנית נחשב כארגומנט וכלול בספירה). הפרמטר השני, argv, הינו מערך של מחרוזות תווים, שכל אחת מהן מתאימה לארגומנט בשורת הפקודה. כדי לגשת לארגומנטי שורת הפקודה יש לכתוב את כותרת הפונקציה main באופן הבא (ניתן כמובן, להחליף את השמות argc ו-argv בשמות אחרים, אך נוהג זה אינו מומלץ, מכיון ששמות אלה מקובלים בספרות כמינוח של ++C):

```
void main(int argc, char *argv[])
```

התקדמות בלולאה עד ש- argv יהיה NULL

כזכור, תוכניות ++C משתמשות בתו NULL (יש לו קוד ASCII) כדי לציין את סיומה של מחרוזת תווים. באופן דומה משתמשת ++C בתו זה כדי לסמן את הכניסה האחרונה במערך argv. התוכנית הבאה, **ARGVNULL.CPP**, דומה לתוכנית הקודמת, אלא שהיא משנה בה את הפקודה for כדי לבצע לולאה על איברי המערך argv, עד אשר היא מזהה את האיבר האחרון, שערכו NULL (שימו לב - אין שימוש ב-argc):

```
#include <iostream.h>

void main(int argc, char *argv[])
  {
   int i;

   for (i = 0; argv[i] != NULL; i++)
     cout << "argv[" << i << "] contains " << argv[i] << endl;
   }
```

הערה 

ביצוע לולאה לסריקת איברי מערך נקראת לעיתים בשם "איטרציה".

argv כמצביע

כפי שלמדנו, C++ מאפשרת לגשת למערכים באמצעות מצביעים. התוכנית הבאה, **ARGVPTR.CPP**, מתייחסת ל-argv כמצביע למצביע מחרוזת תווים (במילים אחרות, מצביע למצביע) ומציגה את ארגומנטי שורת הפקודה באופן הבא:

```
#include <iostream.h>

void main(int argc, char **argv)
 {
  int i = 0;

  while (*argv)
    cout << "argv[" << i++ << "] contains " << *argv++ << endl;
 }
```

בחנו את ההצהרה של הפרמטר argv בכותרת פונקציה main הבאה:

```
void  main(int  argc, char  **argv)
```

הכוכבית הראשונה מציינת למהדר כי argv הינו מצביע, והכוכבית השנייה מציינת כי argv הינו מצביע למצביע - במקרה זה, argv הינו מצביע אל מצביע מסוג char. חישבו על argv כעל מערך של מצביעים, שכל כניסה בו מצביעה למערך של תווים - char (מחרוזת).

שימוש בארגומנטים של שורת הפקודה

התוכנית הבאה, **FILESHOW.CPP**, משתמשת בארגומנטים של שורת הפקודה כדי להציג על המסך את תוכן הקובץ המצוין כארגומנט השני בשורת הפקודה. לדוגמה, הפעלת שורת הפקודה הבאה תגרום להצגת תוכן הקובץ autoexec.bat (שנמצא בספריית השורש) על המסך:

```
C:>FILESHOW \AUTOEXEC.BAT   <Enter>
```

התוכנית **FILESHOW.CPP**, מתחילה בבדיקת ערכו של הפרמטר argc כדי לוודא שהמשתמש אכן ציין קובץ מקור כלשהו בשורת הפקודה. אם שורת הפקודה אכן כוללת שם קובץ, ערכו של הפרמטר argc יהיה 2. לאחר מכן, התוכנית פותחת את הקובץ ומציגה את תוכנו תוך שימוש בטכניקות שנלמדו בפרק 34. אם התוכנית אינה מצליחה לפתוח את הקובץ, היא מציגה הודעת שגיאה ומפסיקה.

```
#include <iostream.h>
#include <fstream.h>
#include <stdlib.h>

void main(int argc, char *argv[])
 {
  char line[256];

  if (argc < 2)
   {
    cerr << "You must specify a filename" << endl;
    exit(1);
   }

  ifstream input_file(argv[1]);

  if (input_file.fail())
    cerr << "Error opening BOOKINFO.DAT" << endl;
  else
   {
    while ((! input_file.eof()) && (!input_file.fail()))
       {
         input_file.getline(line, sizeof(line));
         if (! input_file.fail())
          cout << line << endl;
       }
   }
 }
```

גישה למשתני הסביבה של מערכת ההפעלה

רוב מערכות ההפעלה מאפשרות להגדיר **משתני סביבה** (Environment settings) שתוכניות יכולות לגשת אליהם כדי לנצל ואף לעדכן משתני סביבה שונים. משתנה כזה, לדוגמה הוא נתיב הפקודה (Command path). בסביבת MS-DOS, למשל, יש להשתמש בפקודה set כדי לקבוע או להציג משתני סביבה. כתלות בסוג המהדר שברשותך, ייתכן שבאפשרותך לגשת למשתני הסביבה מתוך תוכניותיך על ידי שימוש בפרמטר שלישי לפונקציה main, פרמטר env. כמו הפרמטר argv, גם פרמטר זה הינו מערך של מחרוזות המסתיים בכניסה NULL. אם ברשותך מהדר התומך בפרמטר env, תוכל לכתוב את כותרת הפונקציה main באופן הבא:

```
void main(int argc, char *argv[], char *env[])
```

התוכנית הבאה, **SHOWENV.CPP**, סורקת בלולאה את איברי המערך env, כדי להציג את משתני הסביבה של התוכנית:

```
#include <iostream.h>

void main(int argc, char *argv[],char *anv[])
 {
   while (*anv)
     cout << *anv++ << endl;
  }
```

התוכנית מתקדמת בלולאה על פני כניסות המערך env עד שהיא מגיעה לערך NULL (הכניסה האחרונה במערך). כאשר מהדרים ומריצים את התוכנית, יוצגו משתני הסביבה הקיימים במחשב, כמו למשל:

```
C:\> SHOWENV <Enter>
TEMP=C:\WINDOWS\TEMP
PROMPT=$p$g
COMSPEC=C:\WINDOWS\COMMAND.COM
PATH=C:\WINDOWS;C:\DOS
```

גישה למשתני הסביבה

כתלות בסוג המהדר שברשותך, התוכניות שתריץ תוכלנה לגשת להגדרות הסביבה של מערכת ההפעלה תוך שימוש בפרמטר שלישי, env, לפונקציה main. כמו הפרמטר argv, גם env הוא מערך של מצביעים למחרוזות המצביעים בהתאמה על הגדרות (פרמטרים) של הסביבה שנמצאות באיברי המערך. כדי לגשת אל הגדרות הסביבה האלו תוך שימוש בפרמטר env, יש לכתוב את כותרת הפונקציה main באופן הבא:

```
void main(int argc, char *argv[], char *env[])
```

סיכום

כדי להעשיר את מיגוון המטלות שתוכניותיך מסוגלות לבצע, ++C מאפשרת להן לגשת ולנצל את הארגומנטים של שורת הפקודה.

בפרק 37 נלמד כיצד יכולות פקודות מאקרו וקבועי שמות לפשט את התכנות ולהפיק קוד קריא וברור יותר.

לפני שנמשיך, הבה נבחן את ידיעותינו:

- ✓ כאשר מריצים תוכנית דרך שורת הפקודה (System prompt) של DOS, הנתונים שהוקלדו אחרי שם התוכנית הופכים לארגומנטים של התוכנית.
- ✓ כדי לאפשר לתוכנית לגשת לארגומנטים של שורת הפקודה, ++C מעבירה שני פרמטרים, argc ו-argv, אל הפונקציה main.

✓ הפרמטר argc מכיל את מספר הארגומנטים בשורת הפקודה.

✓ הפרמטר argv הינו מערך של מחרוזות, אשר כל אחת מהן מכילה בהתאמה ארגומנט של שורת פקודה.

✓ כתלות במהדר שברשותך, התוכניות שתריץ תוכלנה להוסיף לפונקציה main את הפרמטר env, שהינו מערך של מחרוזות המכילות את הגדרות הסביבה.

תרגילים

1. כיתבו את הפונקציה copy: התוכנית מקבלת שני פרמטרים של שורת הרצה (שני שמות קבצים) ומעתיקה את תוכן הקובץ הראשון (אם אינו קיים, היא מודיעה על שגיאה) אל הקובץ השני.
2. כיתבו את הפונקציה grep: התוכנית מקבלת כפרמטרים בשורת ההרצה שם של קובץ ומחרוזות. התוכנית תדפיס את כל שורות הקובץ המכילות לפחות את אחת מהמחרוזות, ותציין עבור כל שורה כזו את מספרה הסידורי בקובץ.
3. כיתבו תוכנית המדפיסה את env ואת argv, אבל בסדר הפוך לסדר קליטתן (מהסוף להתחלה).
4. כיתבו תוכנית המקבלת פקודת DOS ושמות של קבצים, כפרמטרים בשורת הרצה, ותפעיל את הפקודה על כל אחד מהקבצים.

הערה: *מיצאו והיעזרו בפונקציית הספריה s.*

פרק 37

שימוש בקבועים ובהוראות מאקרו

כדי לסייע למתכנת לכתוב תוכניות קריאות יותר, C++ תומכת בשימוש בקבועים בעלי שמות (Named constants) ובהוראות מאקרו (Macro directives). השימוש בקבועי שמות בקוד המקור מאפשר להחליף ערך מספרי, כמו 50, בקבוע שם בעל משמעות, כמו "גודל-כיתה", CLASS_SIZE. כאשר מתכנת אחר יקרא את התוכנית, לא יהיה עליו לנחש כל פעם מהי משמעות הערך המספרי 50 שהוא רואה כרגע בקוד. בכל פעם ש"ייתקל" בקבוע CLASS_SIZE, יהיה לו ברור, שקבוע זה מכיל ערך המתאים למספר התלמידים בכיתה. באופן דומה, על ידי שימוש בהוראת מאקרו, ניתן להחליף שימוש מסורבל במשוואות מורכבות כמו:

```
result = (x*y-3) * (x*y-3) * (x*y-3);
```

בהוראות מאקרו CUBE, כפי שנראה להלן (תחביר השימוש במאקרו דומה לזה של קריאה לפונקציה):

```
result = cube(x*y-3);
```

במקרה זה, המאקרו לא רק משפר את קריאות הקוד, אלא גם מפשט את המשפט ובכך מפחית את הסיכוי לטעות. בהמשך הפרק נדון בנושאים הבאים:

- כדי להפוך תוכניות לקריאות יותר, נהוג ורצוי להחליף ערכים מספריים בקבועי שמות בעלי משמעות.
- השימוש בקבועי שמות במקום בערכים מספריים מקל על עריכת שינויים ועל הטיפול בתוכנית.
- C++ מאפשרת לפשט את כתיבתן של משוואות מורכבות, על ידי שימוש בשמות מאקרו בעלי משמעות.
- לפני הידור התוכנית, מפעיל מהדר C++ תוכנית מיוחדת הנקראת **קדם-מעבד** (Preprocessor). תוכנית זו מחליפה כל משתנה שם והוראת מאקרו בערך המתאים להם (ערך מספרי למשל, או ביטוי מתימטי, בהתאמה).
- הוראות מאקרו מתבצעות מהר יותר מאשר קריאות לפונקציות, אך לעומת זאת, הן גורמות לגידול באורך הקוד (התוכנית בשפת מכונה).
- רוב מהדרי C++ מגדירים מראש קבועי שמות והוראות מאקרו שבהם ניתן להשתמש בגוף התוכנית.

השימוש בקבועי שמות

קבוע שם (Named constant) הוא אוסף תווים חוקי, המקבל ערך קבוע, ואשר בניגוד לערכו של משתנה, אינו יכול להשתנות בעת ריצת התוכנית. ניתן ליצור קבוע שם על ידי שימוש בהוראה **define**#. הוראה זו מיוחדת המיועדת עבור קדם-המעבד של המהדר. לדוגמה, המשפט הבא מגדיר את קבוע השם CLASS_SIZE כבעל ערך 50 :

```
#define  CLASS_SIZE  50
```

כדי להבדיל בין קבוע שם לבין משתנה, נהוג להשתמש באותיות גדולות (Uppercase) לתיאור קבועי שמות. התוכנית הבאה, **constant.cpp**, מגדירה ומציגה את קבוע השם CLASS_SIZE.

```
#include <iostream.h>

#define CLASS_SIZE 50 // Number of students in the class

void main(void)
 {
  cout << "CLASS_SIZE constant is " << CLASS_SIZE << endl;
 }
```

כפי שניתן לראות, הגדרת הקבוע נעשית על ידי שימוש בהוראה define# שנמצאת סמוך לתחילת התוכנית. לאחר הגדרת הקבוע, ניתן להשתמש בערכו במהלך התוכנית על ידי התייחסות לשמו.

הערה



שים לב שהגדרת הקבוע לא מסתיימת בנקודה-פסיק. כתיבת נקודה-פסיק בסוף ההגדרה, תגרום לקדם-המעבד לכלול את התו נקודה-פסיק בתוך ההגדרה. לדוגמה, אם כותבים נקודה-פסיק אחרי הערך **50** בהוראה define# של התוכנית **CONSTANT.CPP**, קדם-המעבד יחליף כל הופעה של הקבוע CLASS_SIZE בערך **;50** (הערך **50** ולאחריו נקודה-פסיק). קרוב לוודאי, שהחלפה זו תגרום לשגיאת תחביר.

הבנת הגדרות קדם-המעבד

כשלב ראשון בהידור התוכנית מריץ המהדר תוכנית מיוחדת הנקראת קדם-מעבד (Preprocessor) קדם-המעבד בודק אילו שורות בתוכנית מתחילות בתו # (למשל #define, או #include). כאשר לדוגמה, תוכנית קדם-המעבד נתקלת בהוראה #include, היא מכניסה את הקובץ הכתוב בהוראה זו אל קובץ המקור של התוכנית, כאילו התוכן שלו הוקלד ישירות לתוך קוד המקור (שים לב שכל התוכניות המופיעות בספר זה השתמשו בהוראה #include, כדי להורות למעבד הראשוני לכלול את תוכן הקובץ iostream.h בקובץ המקור).

כאשר קדם-המעבד נתקל בהוראה #define הוא יוצר קבוע שם, או מאקרו. כאשר הוא נתקל בשם קבוע או בשם מאקרו במהלך סריקת התוכנית, הוא מחליף את השם בערך המוגדר בהוראה #define המתאימה.

כצפוי, הגדרת קבועים אינה מוגבלת לערכים מספריים שלמים בלבד. ניתן להשתמש בקבועים המייצגים מחרוזות ומספרים בנקודה צפה. למשל, התוכנית הבאה, **BOOKINFO.CPP**, משתמשת בהוראות #define כדי ליצור שלושה קבועים, המכילים מידע אודות ספר זה (במקור באנגלית):

```
#include <iostream.h>

#define TITLE "Rescued by C++, Second Edition"
#define LESSON 37
#define PRICE 22.95

void main(void)
  {
    cout << "Book Title: " << TITLE << endl;
    cout << "Current Lesson: " << LESSON << endl;
    cout << "Price: $" << PRICE << endl;
  }
```

כאשר מהדרים ומריצים תוכנית זו, הפלט הבא מופיע על המסך:

```
C:\> BOOKINFO  <Enter>
Book Title: Rescued by C++, Second Edition
Current Lesson: 37
Price" $22.95
```

יצירת קבועי שמות בעזרת define#

נוהל מומלץ ומקובל מאוד לשיפור קריאות ובהירות של תוכניות ++C הוא החלפת ערך מסוים, העובר לאורך התוכנית, בקבוע שם בעל משמעות, המייצג את הערך הזה. כדי להגדיר קבוע שם, משתמשים בהוראה define#. יש לכתוב את ההגדרה הזו בסמוך לראש התוכנית (מכיון שהקבוע נכנס לתוקף מרגע הגדרתו). כמו כן, כדי להבדיל בין קבועים למשתנים, נהוג להשתמש באותיות גדולות עבור שמות קבועים. למשל, ההוראה define# הבאה יוצרת קבוע בשם SECONDS_PER_HOUR :

```
#define  SECONDS_PER_HOUR  3600
```

כשלב ראשון בהידור התוכנית, קדם-המעבד מחליף כל הופעה של השם המשמעותי SECONDS_PER_HOUR בערך המספרי 3600.

חשוב לשים לב שהגדרת הקבוע אינה מסתיימת בנקודה-פסיק. אם יופיע התו נקודה-פסיק לאחר המספר 3600, הוא יובן כחלק מהערך להחלפה, כלומר - '3600;', וקרוב לוודאי, שהחלפה זו תגרום לשגיאת תחביר.

קבועי שמות מקלים על הכנסת שינויים בתוכנית

השימוש בקבועי שמות לא רק תורם לשיפור קריאות התוכנית, אלא גם הופך אותה לקלה וגמישה יותר לשינוי. למשל, כפי שניתן לראות, קטע התוכנית הבא משתמש מספר פעמים בערך 50 (בדוגמה זו, מספר התלמידים בכיתה) :

```
#include <iostream.h>

void main(void)
  {
    int test_scores[50];
    char grades[50];

    int student;

    for (student = 0; student < 50; student++)
      get_test_score(student);

    for (student = 0; student < 50; student++)
      calculate_grade(student);

    for (student = 0; student < 50; student++)
      price_grade(student);
  }
```

כעת, נניח לדוגמה, שמספר התלמידים בכיתה גדל לפתע ל- 55. שינוי זה מחייב החלפת המספר 50 במספר 55 **בכל** מקום בתוכנית שבו הוא מוזכר. התוכנית הבאה, **CLASS.CPP**, ממחישה כיצד אפשר להימנע מעבודה מייגעת ומיותרת זו (העלולה לגרום לטעויות ובלבול, אם אינה מתבצעת בקפדנות ובדיוק), על ידי שימוש בקבוע השם CLASS_SIZE.

הערה 

תוכנית זו אינה שלמה ומטרתה להדגים הוראת מאקרו בלבד. כאשר תריצו את התוכנית תוצגנה הודעות שגיאה שאינן מפריעות להבנת התוכנית.

```
#include <iostream.h>

#define CLASS_SIZE 50

void main(void)
  {
    int test_scores[CLASS_SIZE];
    char grades[CLASS_SIZE];

    int student;

    for (student = 0; student < CLASS_SIZE; student++)
      get_test_score(student);

    for (student = 0; student < CLASS_SIZE; student++)
      calculate_grade(student);

    for (student = 0; student < CLASS_SIZE; student++)
      price_grade(student);
  }
```

בתוכנית זו ראינו, שכדי לשנות את מספר התלמידים בכיתה, צריך לשנות **רק** את הקבוע המוגדר בהוראה #define:

```
#define  CLASS_SOZE  55
```

החלפת משוואות בהוראות מאקרו

כאשר תוכניות מבצעות פעולות מעשיות (ולא תרגול), הן כוללות לעיתים קרובות גם ביטויים ארוכים מסובכים, כמו (ויותר...):

```
result = (x*y-3) * (x*y-3) * (x*y-3);
```

במקרה זה, התוכנית מחשבת את החזקה השלישית של הביטוי (x*y-3). כדי שהתוכנית תהיה ברורה ופשוטה יותר, וכדי להפחית את הסיכוי לטעות (ולהתבלבל) במהלך הקלדת הביטוי, ניתן ליצור מאקרו בשם CUBE ולהשתמש בו באופן הבא:

```
result = CUBE(x*y-3);
```

את המאקרו מגדירים (או יוצרים) באמצעות ההוראה define#. שים לב, שכדי להבדיל בין הוראות מאקרו לפונקציות, נהוג להשתמש באותיות גדולות עבור שמות מאקרו.

המשפט הבא יוצר, או מגדיר, את המאקרו CUBE:

```
#define  CUBE(x)  ((x)*(x)*(x))
```

כפי שניתן לראות, התוכנית מגדירה את המאקרו CUBE כמכפלת הפרמטר x בעצמו, פעמיים.

התוכנית הבאה, **SHOWCUBE.CPP**, משתמשת במאקרו CUBE כדי להציג את החזקה השלישית של המספרים השלמים בטווח 1 עד 10:

```
#include <iostream.h>

#define CUBE(x) ((x)*(x)*(x))

void main(void)
 {
  for (int i = 1; i <= 10; i++)
      cout << i << " cubed is " << CUBE(i) << endl;
 }
```

בזמן הידור התוכנית, קדם המעבד מחליף כל הופעה של CUBE בביטוי המתאים (על-פי הגדרת המאקרו). במילים אחרות, כך ייראה הקוד של התוכנית **SHOWCUBN.CPP**, אחרי פעולת קדם-המעבד:

```
#include <iostream.h>
#define CUBE(x) ((x)*(x)*(x))
void main(void)
 {
  for (int i = 1; i <= 10; i++)
      cout << i << " cubed is " << ((i)*(i)*(i)) << endl;
 }
```

חשוב לשים לב, שבהגדרת המאקרו CUBE, כל הופעה של הפרמטר x הוקפה בסוגריים, כך ((x) * (x) * (x)) ולא כך: (x * x * x). ככלל, כאשר יוצרים מאקרו, צריך לכתוב את הפרמטרים בין סוגריים, כדי להבטיח כי ++C תעריך את הביטויים באופן הרצוי. כזכור משיעור 5, ++C פועלת לפי סדר קדימויות של האופרטורים כדי לקבוע את סדר ביצוע הפעולות החשבוניות. נניח כעת, כי תוכנית כלשהי משתמשת במאקרו CUBE עם הביטוי 3+5-2 בצורה זו:

```
result = CUBE(3+5-2);
```

אם הגדרת המאקרו מקבצת את הפרמטרים בתוך סוגריים, קדם-המעבד יצור את המשפט הבא:

```
result = ((3+5-2) * (3+5-2) * (3+5-2));
```

אבל, אם בהגדרת המאקרו מושמטים הסוגריים, קדם-המעבד ייצור את המשפט הבא:

```
result = (3+5-2*3+5-2*3+5-2);
```

אם נחשב את שני הביטויים, נמצא כמובן כי הם **שונים** זה מזה, והביטוי השני אינו משקף את מה שאנו התכוונו לו. כלומר, על ידי קיבוץ פרמטרי מאקרו בתוך סוגריים ניתן למנוע שגיאות מסוג זה.

ההבדל בין מאקרו לפונקציה

הגדרת מאקרו היא אינה פונקציה כאשר תוכנית משתמשת בפונקציה, רק העתק אחד של משפטי הפונקציה מצוי בקוד המכונה של התוכנית. בזמן הריצה, בכל פעם שהתוכנית קוראת לפונקציה, היא דוחפת (Push) פרמטרים לתוך המחסנית ואז מדלגת אל קטע התוכנית של הפונקציה. כאשר הפונקציה מסיימת, התוכנית מסלקת ערכים שהוכנסו למחסנית לפני הקריאה לפונקציה ומדלגת חזרה אל המשפט (הפקודה, או ההוראה) בתוכנית, שנמצא מייד אחרי משפט הקריאה לפונקציה.

עם מאקרו לעומת זאת (כמו במקרה של הגדרת קבוע שם), המצב שונה. קדם-המעבד מחליף כל התייחסות למאקרו בגוף הקוד בהגדרת המאקרו המתאימה, שנכתבה בראש התוכנית, בדרך כלל. למשל, אם התוכנית הקודמת היתה משתמשת במאקרו CUBE במאה מקומות שונים, קדם-המעבד היה "שותל" את הקוד המתאים של המאקרו 100 פעמים שונות. עם זאת, השימוש במאקרו חוסך את תקורת הפקודות הדרושות לקריאה לפונקציה (התקורה מתקבלת מהצורך לדחוף ולסלק פרמטרים מהמחסנית ומהצורך לקפוץ לקוד הפונקציה ולחזור ממנה). הסיבה לכך היא, שכל שימוש בהוראת מאקרו בגוף התוכנית, מוחלף על ידי קדם-המעבד בגוף המאקרו (המתקבל על פי הגדרת המאקרו והארגומנטים האקטואליים בעת השימוש בו). זהו שילוב של קוד המאקרו במלואו (Inline) בגוף התוכנית, בכל מקום שבו יש שימוש במאקרו. על כן, השימוש בהוראות מאקרו מגדיל את נפח קוד המכונה של התוכנית (הקוד הנוצר כתוצאה מהידור התוכנית המתקבלת לאחר פעולת קדם-המעבד).

השימוש בהוראת מאקרו גמיש ביותר

השימושים בהוראות מאקרו מגוונים ביותר. יחד עם זאת יש לזכור, כי מטרת השימוש במאקרו היא פישוט קוד התוכנית ושיפור קריאותו. התוכנית הבאה, **MACDELAY.CPP**, ממחישה את הגמישות שבשימוש במאקרו. בנוסף לכך, היא גם יכולה לתת תמונה טובה יותר כיצד המאקרו מוחלף באוסף המשפטים המתאים על פי ההגדרה.

```
#include <iostream.h>

#define delay(x) {\
                  cout << "Delaying for " << x << endl; \
                  for (long int i = 0; i < x; i++) \
                    ; \
                  }

void main(void)
 {
  delay(100000L);

  delay(200000L);

  delay(300000L);
 }
```

במקרה זה, מכיון שהגדרת המאקרו נמשכת לאורך מספר שורות, יש להוסיף לוכסן הפוך (\) בסוף כל שורה, אשר אינה השורה האחרונה בהגדרת המאקרו. כאשר קדם-המעבד פוגש התייחסות למאקרו, הוא ישתול במקומה את כל המשפטים המופיעים בהגדרת המאקרו.

סיכום

הוראות מאקרו וקבועי שמות יכולים לשפר את קריאות התוכנית ובדרך זו - לפשט את התכנות. בפרק זה ראינו כיצד ליצור ולהשתמש בקבועי שמות ובהוראות מאקרו בתוך הקוד שאנו כותבים.

בפרק 38 נלמד על פולימורפיזם (Polymorphism) – שיטה אשר מאפשרת לעצם (Object) לשנות צורה במהלך ריצת התוכנית.

אך לפני שנמשיך, הבה נחזור על עיקרי הדברים שלמדנו בפרק זה:

- ✓ הוראות מאקרו וקבועי שמות משפרים את קריאות התוכנית על ידי החלפת משוואות מסובכות וקבועים מספריים בשמות בעלי משמעות.
- ✓ על ידי החלפת ערך מספרי בקבוע שם לאורך התוכנית, קטן מספר השינויים אשר יש לבצע מאוחר יותר, כאשר ערך הקבוע משתנה.

✓ במהלך הידור תוכנית, מהדר ++C מפעיל תוכנית מיוחדת, הנקראת קדם-מעבד (Preprocessor), המחליפה כל קבוע שם או מאקרו בערך המתאים לו.

✓ השימוש במאקרו מהיר יותר מאשר קריאה לפונקציה. כל שימוש בהוראת מאקרו מגדיל את קוד המכונה המתקבל.

✓ כאשר אורך הגדרת מאקרו עולה על שורה אחת, ניתן לכתוב לוכסן הפוך (\) בסוף השורה, אשר מסמן לקדם-המעבד שההגדרה ממשיכה בשורה הבאה.

תרגילים

1. כיתבו תוכנית המגדירה קבוע nb_chars ואז קוראת מהקלט מספר כזה של תווים (אחרי הקלדת כל תו מקישים על Enter). לבסוף, התוכנית מדפיסה את המחרוזת שהם יוצרים.

2. כיתבו מאקרו בשם AVG המקבל שני ארגומנטים ומחזיר את הממוצע שלהם. שימו לב שהמאקרו הוא ללא הגדרת סוג (typeless), ולכן הוא יכול לפעול עם ארגומנטים מטיפוסים נומריים שונים. בידקו זאת.

3. מהו הפלט של התוכנית הבאה? שימו לב להבדל בין השימוש במאקרו לבין שימוש בפונקציה (או בפונקציה משולבת).

```
#include <iostream.h>

#define H cout << G << endl

int G = 5;

void P(void) { cout << G << endl; }

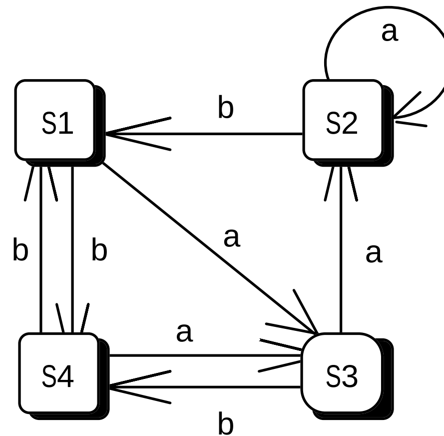
inline void M(void)  {   cout << G << endl;   }

void F1(int G = 7) {   P();   }
void F2(int G = 7) {   M();   }
void F3(int G = 7) {   H;   }

void main(void)
  {
   F1();   F2();   F3();
  }
```

4. כיתבו תוכנית למכונת המצבים (אוטומט סופי) הבאה: המכונה פועלת מעל האלף-בית {a,b}. יש בה ארבעה מצבים: S1, S2, S3, ו-S4. המצב ההתחלתי הוא S1. כל מצב יוגדר בעזרת ההוראה #define.

כאשר המשתמש מקיש על תו שאינו שייך לאלף-בית של המכונה, מסתיימת הריצה (המכונה עוצרת).



פרק 38

פולימורפיזם

כאשר מדברים על ++C ועל **תכנות מונחה עצמים** - OOP (**O**bject **O**riented **P**rogramming) משתמשים הרבה במונח **פולימורפיזם** (Polymorphism). אפשר לומר, ש**פולימורפיזם** הוא יכולתו של עצם לשנות צורה. אם נפרק את המילה "פולימורפיזם", נראה כי "פולי" משמעותו הרבה ו"מורפיזם" מתייחס לשינוי צורה. **עצם פולימורפי**, אם כן, הוא עצם, אשר יכול להיות בעל צורות רבות. בפרק זה נלמד מהו פולימורפיזם וכיצד להשתמש בעצמים פולימורפיים בתוכניות, כדי לפשט ולהקטין את כמות הקוד שכותבים.

במהלך הפרק נדון בנושאים הבאים:

- ❖ פולימורפיזם הוא יכולתו של עצם לשנות את צורתו במהלך ריצת התוכנית.
- ❖ ++C מאפשרת ליצור עצמים פולימורפיים בצורה פשוטה וברורה.
- ❖ כדי ליצור עצמים פולימורפיים, על התוכנית להשתמש בפונקציות וירטואליות.
- ❖ פונקציה וירטואלית היא פונקציית מחלקה בסיסית, אשר לפני שמה יש לכתוב את המילה השמורה **virtual**.
- ❖ כל מחלקה הנגזרת ממחלקה בסיסית (Base class) יכולה להשתמש בפונקציה וירטואלית ולהעמיס (Overload) אותה.
- ❖ כדי ליצור עצם פולימורפי, צריך להשתמש במצביע לעצם מטיפוס המחלקה הבסיסית.

מהו פולימורפיזם

עצם פולימורפי הוא **עצם** שיכול לשנות את צורתו בזמן ריצת התוכנית. נניח למשל, שתוכניתנית העובדת בחברת "בזק" צריכה לכתוב תוכנית, שמדמה פעולות טלפוניות. התוכניתנית יודעת שקיימות מספר פעולות פשוטות לחיוג בטלפון (חיוג, שיחה, ניתוק, פעולה במצב של "תפוס" וכדי). פעולות אלו מאפשרות להגדיר את המחלקה הבאה:

```
class phone  {
 public:
  void dial(char *number) { cout << "Dialing "
                                     << number << endl; }
  void answer(void) { cout << "Waiting to answer call" << endl; }
  void hangup(void){ cout << "Done with call-hanging up"
                                              << endl; }
  void ring(void) { cout << "Ring, ring, ring" << endl;}
  phone(char *number) { strcpy(phone::number, number); };
 private:
  char number[13];
};
```

התוכנית **PHONEONE.CPP**, משתמשת במחלקה **phone** כדי ליצור עצם "טלפון":

```
#include <iostream.h>
#include <string.h>

class phone  {
 public:
  void dial(char *number) { cout << "Dialing "
                              << number << endl; }
  void answer(void) { cout << "Waiting to answer call" << endl; }
  void hangup(void){ cout << "Done with call-hanging up"
                                              << endl; }
  void ring(void) { cout << "Ring, ring, ring" << endl;}
  phone(char *number) { strcpy(phone::number, number);};
 private:
  char number[13];
};

void main(void)
 {
   phone telephone("555-1212");
   telephone.dial("212-555-1212");
 }
```

כאשר אותה עובדת "בזק" מראה את התוכנית למנהלת שלה, היא מציינת (בצער, כמובן), שלא קיימת בתוכנית הפרדה בין פעולה בטלפון חוגה לבין פעולה בטלפון לחצנים, וכי התוכנית לא מספקת תמיכה בטלפונים ציבוריים, בהם צריך לשלם לפני השימוש. מכיון שהתוכניתנית המוכשרת מבינה ויודעת מהי **ירושה**, היא מחליטה **לגזור** (Derive) את המחלקות **touch_tone** ו- **pay_phone** מהמחלקה **phone** כפי שנראה להלן:

```
class touch_tone : phone  {
 public:
  void dial(char *number) { cout << "Beep Beep Dialing "
                                      << number << endl; }
  touch_tone(char *number) : phone (number) { }
 };

class pay_phone : phone  {
 public:
  void dial(char *number) { cout << "Please deposit "
                          << amount << " cents" << endl; }
                  cout << "Dialing " << number <<  endl; }
  pay_phone(char *number, int amount) : phone(number) {
                             pay_phone::amount = amount; }
 private:
  int amount;
 };
```

כפי שניתן לראות, המחלקות touch_tone ו- pay_phone מגדירות פונקציות מחלקה dial משלהן. אם מניחים, ששיטת החיוג של המחלקה phone מבוססת על טלפון חוגה, אין צורך ליצור מחלקת טלפון חוגה עצמאית. התוכנית הבאה, **NEWPHONE.CPP**, משתמשת במחלקות שהזכרנו, כדי ליצור אובייקטים מהטיפוסים phone, touch_tone ו- pay_phone:

```
#include <iostream.h>
#include <string.h>

class phone  {
 public:
  void dial(char *number) { cout << "Dialing "
                                  << number << endl; }
  void answer(void) { cout << "Waiting to answer call" << endl; }
  void hangup(void){ cout << "Done with call-hanging up"
                                                << endl; }
  void ring(void) { cout << "Ring, ring, ring" << endl;}
  phone(char *number) { strcpy(phone::number, number);};
 protected:
  char number[13];
};
```

```
class touch_tone : phone  {
 public:
  void dial(char *number) { cout << "Beep Beep Dialing "
                                      << number << endl; }
  touch_tone(char *number) : phone (number) { }
 };

class pay_phone : phone  {
 public:
  void dial(char *number) { cout << "Please deposit "
                      << amount << " cents" << endl;
                 cout << "Dialing " << number<<  endl; }
  pay_phone(char *number, int amount) : phone(number) {
                         pay_phone::amount = amount; }
 private:
  int amount;
 };

void main(void)
 {
  phone rotary("303-555-1212");
  rotary.dial("602-555-1212");

  touch_tone telephon("555-1212");
  telephon.dial("212-555-1212");

  pay_phone city_phone("555-1111", 25);
  city_phone.dial("212-555-1212");
 }
```

כאשר מהדרים ומריצים את התוכנית, יופיע הפלט הבא על המסך:

```
C:\>NEWPHONE  <Enter>
Dialing 602-555-1212
Beep Beep Dialing 212-555-1212
Please deposit 25 cents
Dialing 212-555-1212
```

כאמור, עצם פולימורפי הוא עצם אשר **משנה את צורתו** תוך כדי ריצת התוכנית. התוכנית הקודמת, למשל, לא השתמשה בעצמים פולימורפיים. ובמילים אחרות, אף עצם לא שינה בה את צורתו.

יצירת עצם phone פולימורפי

לאחר שתוכניתנית "בזק" הראתה למנהלת שלה את התוכנית החדשה שפיתחה, אמרה המעבידה שעצם הטלפון חייב, על פי דרישה, לאפשר סימולציה של טלפון חוגה, טלפון לחצנים וגם טלפון ציבורי. במילים אחרות, בשיחה אחת יכול עצם הטלפון להיות טלפון חוגה, אך בשיחה הבאה אולי יהיה טלפון לחצנים, ובשיחה נוספת הוא עשוי להיות טלפון ציבורי, וכן הלאה. כלומר, משיחת טלפון אחת לבאה אחריה, על עצם הטלפון לשנות צורה.

אם נתבונן בתוכנית, נראה, שבין המחלקות יש שוני בולט, והוא - פונקציית החיוג, **dial**. כדי ליצור עצם פולימורפי, צריך קודם כל להפוך את פונקציות מחלקת הבסיס, הנבדלות מהפונקציות הנגזרות, ל**פונקציות וירטואליות** (Virtual functions) עושים זאת על ידי כתיבת המילה השמורה **virtual** לפני כתיבת ההצהרה של הגדרת הפונקציה, כפי שנראה להלן:

```
class phone  {
 public:
  virtual void dial(char *number) { cout << "Dialing "
                                    << number << endl;}
  void answer(void) { cout << "Waiting to answer call" << endl;}
  void hangup(void){ cout << "Done with call-hanging up"
                                              << endl;}
  void ring(void) { cout << "Ring, ring, ring" << endl;}
  phone(char *number) { strcpy(phone::number, number);};
 protected:
  char number[13];
};
```

כעת, יש ליצור בתוכנית מצביע לאובייקט מטיפוס מחלקת בסיס. בתוכנית הטלפון הזו, ניצור מצביע אל המחלקה הבסיסית phone:

```
phone *poly_phone;
```

כדי לשנות את צורת העצם, יש להציב במצביע כתובת של עצם מטיפוס המחלקה הנגזרת ממחלקת הבסיס, כפי שנראה להלן:

```
poly_phone = (phone *) &home_phone;
```

צורת הכתיבה (* phone), לאחר אופרטור ההשמה הינו cast, אשר מאשר למהדר ++C להציב את הכתובת של משתנה מסוג אחד (touch_tone) במצביע למשתנה מסוג אחר (phone). מכיון שהתוכנית יכולה להציב כתובות של עצמים שונים ב- poly_phone, העצם יכול לשנות את צורתו ולכן הוא פולימורפי. התוכנית הבאה, **POLYMORP.CPP**, משתמשת בטכניקות אלו כדי ליצור עצם טלפון פולימורפי (למעשה, המצביע הוא הישות הפולימורפית. הוא יכול להצביע על כל עצם מטיפוס מחלקה נגזרת כלשהי. אף על פי כן, נגיד כי העצם poly_phone הוא פולימורפי, ולא נבחין

מפורשות בעובדה שזהו למעשה מצביע פולימורפי). בזמן ריצת התוכנית, העצם poly_phone משתנה מטלפון חוגה לטלפון לחצנים ולטלפון ציבורי:

```
#include <iostream.h>
#include <string.h>

class phone  {
 public:
  virtual void dial(char *number) { cout << "Dialing "
                                    << number << endl; }
  void answer(void) { cout << "Waiting to answer call" << endl; }
  void hangup(void){ cout << "Done with call-hanging up"
                                              << endl; }
  void ring(void) { cout << "Ring, ring, ring" << endl;}
  phone(char *number) { strcpy(phone::number, number);};
 protected:
  char number[13];
};

class touch_tone : phone  {
 public:
  void dial(char *number) { cout << "Beep Beep Dialing "
                                    << number << endl; }
  touch_tone(char *number) : phone (number) { }
 };

class pay_phone : phone  {
 public:
  void dial(char *number) { cout << "Please deposit "
                       << amount << " cents" << endl;
                cout << "Dialing " << number; }
  pay_phone(char *number, int amount) : phone(number) {
                          pay_phone::amount = amount; }
 private:
  int amount;
 };

void main(void)
 {
  pay_phone city_phone("702-555-1212", 25);
  touch_tone home_phone("555-1212");
  phone rotary("201-555-1212");
```

```
  // Make the object a rotary phone
  phone *poly_phone = &rotary;
  poly_phone->dial("818-555-1212");

  // Change object's from to a touchtone phone
  poly_phone = (phone *) &home_phone;
  poly_phone->dial("303-555-1212");

   // Change the object's from to a payphone
  poly_phone = (phone *) &city_phone;
  poly_phone->dial("212-555-1212");
 }
```

כאשר מהדרים ומריצים את התוכנית, יופיע הפלט הבא על המסך :

```
C:\> POLYMORP  <Enter>
Dialing 818-555-1212
Beep Beep Dialing 303-555-1212
Please deposit 25 cents
Dialing 212-555-1212
```

העצם poly_phone משנה את צורתו בזמן ריצת התוכנית, ולכן הוא **עצם פולימורפי**.

עצמים פולימורפיים יכולים לשנות את צורתם בזמן ריצת התוכנית

עצם פולימורפי הוא עצם, אשר יכול לשנות את צורתו במהלך ריצת התוכנית. כדי ליצור עצם פולימורפי, משתמשים במצביע לעצם מטיפוס מחלקת בסיס. לאחר מכן, התוכנית יכולה להציב במצביע את כתובתו של עצם כלשהו מטיפוס נגזר מתאים (טיפוס מחלקה נגזרת ממחלקת הבסיס). כל הצבה כזו משנה את "צורתו" של העצם המצביע (שהינו פולימורפי). אבן היסוד לבניית מצביעים פולימורפיים היא השימוש בפונקציות וירטואליות במחלקת הבסיס.

פונקציות וירטואליות טהורות

כפי שלמדנו, כדי לאפשר עבודה עם עצמים (מצביעים) פולימורפיים, צריך להגדיר פונקציה אחת או יותר של מחלקת הבסיס כפונקציות וירטואליות. המחלקות הנגזרות יכולות להשתמש בפונקציות הווירטואליות הללו, או לספק פונקציות משלהן, אשר תתבצענה במקום (Overiding) הפונקציות הווירטואליות של מחלקת הבסיס. יש מקרים, בהם לא ניתן (או אין שום טעם) לממש פונקציה וירטואלית במחלקת הבסיס. למשל, כאשר כל מחלקה נגזרת תצטרך ממילא לממש פונקציה הייחודית רק לה. במקרים כאלה, בהם ניתן להצהיר על הפונקציה הווירטואלית בלבד, אך אין רוצים לנצל אותה, משתמשים ב**פונקציה וירטואלית טהורה** (Pure virtual function) בפונקציה כזו אין משפטי ביצוע כלשהם.

כדי ליצור פונקציה וירטואלית טהורה, על התוכנית לציין רק את כותרת הפונקציה ו"להציב" בה את הערך אפס, כפי שנראה להלן:

```
class phone  {
 public:
  virtual void dial(char *number) =0 // Pure virtual
                                        function
  void answer(void) { cout << "Waiting to answer call" << endl; }
  void hangup(void){ cout << "Done with call-hanging up"
                                              << endl; }
  void ring(void) { cout << "Ring, ring, ring" << endl;}
  phone(char *number) { strcpy(phone::number, number);};
 protected:
  char number[13];
};
```

כל מחלקה נגזרת, חייבת להגדיר פונקציה עבור כל פונקציה וירטואלית טהורה של מחלקת הבסיס. אם מחלקה נגזרת משמיטה את הגדרת הפונקציה המתאימה לפונקציה וירטואלית טהורה, מהדר ++C יודיע על שגיאת תחביר.

מתי להשתמש בפולימורפיזם

כאשר נגלה שיש צורך לשכפל קוד, כשברצוננו ליצור אלגוריתם לשימוש כללי – נפרק את האלגוריתם לפעולות קטנות. כל פעולה תוגדר כפונקציה וירטואלית. פולימורפיזם מאפשר תכנות נוח, יעילות ונוחות תחזוקה לכן מומלץ לשקול את אפשרות השימוש בו. אם כי יש לקחת בחשבון את מחיר הזמן כלומר, הקישור לפונקציה יתבצע בזמן ריצה (Late Binding). מכיון שלפני ריצת התוכנית אי אפשר לדעת לאיזו פונקציה יש לקשר את הקריאה לפונקציה.

בדוגמה הבאה נראה כיצד פרקנו את הנוסחה לחישוב נפח קוביה ותיבה לפונקציות וירטואליות.

דוגמה : polygnrl.cpp

הערה *: התוכנית נבדקה והורצה תחת Visual C++ 6.0.*

```
#include <iostream.h>

class cube
    {
        protected:
            int a, h;
        public:
            cube(int ai, int hi) { a = ai; h = hi; }
            virtual int area() { return(a * a); }
            int vol(void) { return(h * area()); }
    };

class box: public cube
    {
        protected:
            int b;
        public:
            box(int ci, int di, int i): cube(ci, i) { b= di; }
            int area(void) { return(a * b); }
    };

void main(void)
{
    cube c(5, 8);
    box b(5, 3, 8);
    int rv_cube, rv_box;

    rv_cube = c.vol();
    rv_box = b.vol();
    cout<<"rv_cube = "<<rv_cube<<"\nrv_box = "<<rv_box<<endl;
};
```

מה השינויים שנאלץ לבצע אם נרצה לחשב נפח גליל? פשוט נחשב מחדש את השטח a * a * 3.14 ונשתמש בפונקציה המחשבת נפח קוביה. נדרשת מחשבה רבה מראש להגדרת המבנה הבסיסי של המחלקות שאותם נוריש כבסיס למחלקות חדשות על פי הצרכים העתידיים שאינם ידועים מראש.

סיכום

פולימורפיזם הוא יכולת של עצם לשנות את צורתו במהלך ריצת התוכנית. בפרק זה למדנו את הצעדים הדרושים ליצירת עצמים פולימורפיים. לפני שנמשיך לפרק 39, יש לוודא שהובנו מושגי המפתח והנושאים הבאים:

- ✓ **עצם פולימורפי** יכול לשנות את צורתו במהלך ריצת התוכנית.
- ✓ יוצרים עצמים פולימורפיים על ידי שימוש במחלקות הנגזרות ממחלקת בסיס קיימת.
- ✓ בתוך מחלקת הבסיס של עצם פולימורפי, מגדירים פונקציה אחת או יותר כ**פונקציה וירטואלית**.
- ✓ עצמים פולימורפיים נבדלים זה מזה בשימוש שלהם בפונקציות הווירטואליות של מחלקת הבסיס.
- ✓ כדי ליצור עצם (מצביע) פולימורפי יוצרים מצביע לעצם מחלקת בסיס.
- ✓ כדי לשנות צורה של עצם פולימורפי, מכוונים שהמצביע אליו יופנה אל עצם אחר, על ידי הצבת כתובת העצם האחר במצביע הפולימורפי.
- ✓ פונקציה וירטואלית טהורה היא פונקציה שמחלקת הבסיס אינה מספקת עבורה הגדרה כלשהי. במקום זאת, מחלקת הבסיס "מציבה" בפונקציה את הערך אפס.
- ✓ מחלקות נגזרות חייבות לספק הגדרת פונקציה עבור כל פונקציה וירטואלית טהורה של מחלקת הבסיס.
- ✓ אם במחלקה הנגזרת לא מגדירים מחדש את הפונקציה הווירטואלית, תופעל הפונקציה הווירטואלית השייכת למחלקת הבסיס.

תרגילים

1. נתונה המחלקה המופשטת הבאה:

```
class operation {
  public:
      virtual float result(float num1,float num2) =0;
  };
```

כיתבו מחלקות בשם plus ו-minus היורשות את operation ומפיקות את result. בפונקציה main הגדירו משתנים פולימורפיים op1 ו-op2 מטיפוס *operation ובידקו בעזרתם את המחלקות plus ו-minus שכתבתם.

2. הגדירו מחלקה וירטואלית בשם animal בעלת פונקציה וירטואלית טהורה בשם print, המדפיסה את שם החיה. השם נקבע בעזרת פונקציית הבנייה. יש ליצור מחלקות frog ו-cat היורשות מ-animal ומפעילות אותה. בפונקציה main יש לכלול מצביע פולימורפי שיצביע פעם אל אובייקט מטיפוס cat, ופעם אל אובייקט מטיפוס frog, ויפעיל את שניהם.

3. הרחיבו את התרגיל הקודם לטיפול במערך של חיות ("גן חיות"). בכל צעד ניתן להוסיף לגן החיות באופן אינטראקטיבי, אחד מאלה: (1) צפרדע, או (2) חתול. ניתן גם (3) להדפיס את שמות החיות בגן החיות או (4) לסיים את ריצת התוכנית. המספר המקסימלי של חיות בגן הוא nb_animals.

4. הגדירו מחלקה מופשטת בשם shape בעלת פונקציה וירטואלית טהורה בשם area. המחלקה rectangle תירש את shape ותפעיל פונקציה וירטואלית area. המחלקה square תירש את rectangle, תממש מחדש את area וגם תוסיף פונקציה circumference. בידקו בעזרת הפונקציה main את המחלקות שכתבתם. הסבירו מדוע הפקודה s1->circumference() אינה חוקית.

```
void main(void)
  {
    shape* s1;
    square* s2;

    s1 = new rectangle(4, 7);
    cout << s1->area() << endl;
    s2 = new square(5);
    cout << s2->area() << endl;
    cout << s2->circumference() << endl;
    s1 = s2;
    cout << s1->area() << endl;
    // NOTE THAT THIS IS ILLEGAL:
              cout << s1->circumference() << endl;
  }
```

5. הגדירו מחלקה מופשטת בשם shape בעלת פונקציות וירטואלית טהורות בשם area, circumference, is-in (הבודקת אם הנקודה הנתונה - כארגומנט - נמצאת בתוך הצורה). הגדירו מחלקה position שלה פונקציות set_position ו-move (הזזת הנקודה ב-offsets המסופקים כארגומנטים). המחלקה circle תירש את shape ואת position (שתספק את קואורדינטות מרכז המעגל), תפעיל את הפונקציות הווירטואלית, ותוסיף פונקציה set_radius. בידקו את המחלקות בעזרת פונקציה main מתאימה.

פרק 39

ניצול מצבים חריגים של ++C לטיפול בשגיאות

תוכניתנים מנוסים, אשר מאחוריהם עבר עשיר של כתיבה וניפוי של תוכניות (בדיקתן וסילוק הטעויות מהן), מסוגלים בדרך כלל לאפיין שגיאות טיפוסיות, שעלולות לצוץ בזמן ריצת התוכנית. לדוגמה, תוכנית הקוראת נתונים מתוך קובץ, צריכה לוודא שהקובץ הנתון אכן קיים, וגם ניתן לגשת אליו. בצורה דומה, תוכנית המשתמשת באופרטור new כדי להקצות זיכרון, צריכה לבדוק מצב בו אין מספיק זיכרון למילוי הבקשה, וגם לטפל במצב כזה. מערכות תוכנה גדולות ומורכבות משופעות בבדיקות מעין אלו. בפרק זה, נלמד כיצד להשתמש במנגנון **המצבים החריגים של שפת ++C** (**C++ Exceptions**), המיועד לפשט את תהליך גילוי שגיאות זמן-ריצה ואת הטיפול בהן.

נדון בנקודות המפתח הבאות:

- **מצב חריג** (**Exception**) הוא אירוע לא-צפוי, שגיאה המתרחשת בזמן ריצת התוכנית.
- מצבים חריגים מוגדרים בעזרת מחלקות.
- כדי להנחות את התוכניות לתפוס מצבים חריגים, אנו משתמשים ב**משפט try**.
- כדי לתפוס מצב חריג ספציפי, משתמשים ב**משפט catch**.
- כדי לעורר מצב חריג בשעה שמתגלית טעות, משתמשים ב**משפט throw**.
- כאשר התוכנית מזהה (תופסת) מצב חריג, היא קוראת לפונקציה מיוחדת המטפלת בו. פונקציה זו נקראת **פונקציית טיפול - Exception handler.**
- מהדרים ישנים אינם תומכים במנגנון המצבים החריגים של ++C.

ייצוג מצבים חריגים כמחלקות

מטרת השימוש במנגנון המצבים החריגים של ++C היא פישוט ושיפור של תהליך גילוי שגיאות זמן הריצה (אשר מכנים אותן "מצבים חריגים"), ושל תהליך הטיפול בשגיאות הללו. באופן אידאלי, היינו רוצים שתוכנית הנתקלת בטעות בלתי צפויה, תדע לטפל בה כראוי, ותנסה להתגבר על מצב זה, ולא "סתם" תסיים את הריצה (עם, או בלי הודעת שגיאה מתאימה). בפועל, מצבים חריגים מוגדרים כמחלקות. לדוגמה, המשפטים הבאים מגדירים שלושה מצבים חריגים, הקשורים לטיפול בקבצים:

```
class  file_open_error {};
class  file_read_error {};
class  file_write_error {};
```

בשלב מאוחר יותר של פרק זה, ניצור מצבים חריגים המכילים גם משתני מחלקה ופונקציות מחלקה. לעת עתה, די לנו לדעת שכל מצב חריג מיוצג על ידי מחלקה מתאימה.

כיצד מורים ל- ++C לתפוס מצבים חריגים (משפטי try ו-catch)

כדי לאפשר לתוכנית לתפוס מצב חריג ולטפל בו, יש להשתמש במשפט **try** של ++C. לדוגמה, משפט try הבא מאפשר לתפוס מצבים חריגים בעת ביצוע הפונקציה file_copy:

```
try  {
    file_copy("SOURCE.TXT", "TARGET.TXT");
};
```

מייד לאחר משפט try, צריכים להופיע משפטי catch (אחד או יותר), כדי לקבוע אם התרחש מצב חריג, ואם כן, איזה (**TRY.TXT**):

```
try {
  file_copy("SOURCE.TXT", "TARGET.TXT");
};

catch (file_open_error) {
  cerr << "Error opening the source or target file" <<
endl;
exit(1);
}

catch (file_read_error) {
  cerr << "Error reading the source file" << endl;
exit(1);
}
```

```
catch (file_write_error) {
  cerr << "Error writing the target file" << endl;
exit(1);
}
```

כפי שניתן לראות, בקטע התוכנית הזה, בודקים (באמצעות catch) לאחר הפקודה try אם התעורר אחד המצבים החריגים שהוגדרו קודם לכן. אם התרחש מצב חריג, מודפסת הודעה מתאימה והתוכנית מסיימת את פעולתה. במערכות תוכנה אמיתיות, התגובה רציונלית יותר, כלומר: התוכנית מנסה לתקן את הטעות ולהפעיל שנית את הפונקציה שבעקבות הקריאה לה התעורר המצב החריג.

אם פונקציה מסתיימת בהצלחה (כלומר לא מתעורר מצב חריג), משפטי catch העוקבים אינם משפיעים על מהלך התוכנית.

כיצד לעורר מצב חריג (משפט throw)?

מצבים חריגים אינם מתעוררים מעצמם בזמן ריצת התוכנית. כדי לעורר מצב חריג יש להשתמש במשפט throw. לדוגמה, אם מתגלית שגיאה במהלך ביצוע הפונקציה file_copy (**filecopy.txt**) הפונקציה מעוררת מצב חריג מתאים, שניתן לבדיקה:

```
void file_copy(char *source, char *target)
 {
   char line[256];

   ifstream input_file(source);
   ofstream output_file(target);

   if(input_file.fail())
    throw(file_open_error);
   else if (output_file.fail())
    throw(file_open_error);
   else
    {
      while ((! input_file.eof()) && (! input_file.fail()))
       {
       input_file.getline(line, sizeof(line));
       if(! input_file.fail())
         output_file << line << endl;
       else
          throw(file_read_error);
       if (output_file.fail())
          throw(file_write_error);
      }
    }
  }
```

(שים לב שבקובץ שבתקליטור יש עליך לשנות את { שנמצא אחרי while ל- {)

התוכנית משתמשת במשפט throw כדי לעורר מצבים חריגים שונים, אשר מתאימים לשגיאה שקרתה במהלך הריצה.

פעולת מנגנון המצבים החריגים

כאשר מנצלים את מנגנון המצבים החריגים, התוכניות בודקות שגיאות, ובמידה שדרוש, הן מעוררות מצב חריג (Exception) על ידי משפט throw. הפעלת משפט זה גוררת הפעלת פונקציה מיוחדת, המיועדת לטיפול במצב חריג זה -Exception handler (פונקציה זו מוגדרת בתוך מחלקת המצב החריג). לאחר שפונקציית הטיפול מסיימת את פעולתה, התוכנית ממשיכה לרוץ מהשורה הראשונה שאחרי משפט try. משפט try הוא זה שמאפשר לאתר ולתפוס מצב חריג. כעת, בעזרת משפטי catch מתאימים, התוכנית יכולה לתפוס את המצב החריג המסוים שהתעורר ולפעול בהתאם.

הגדרת פונקציה לטיפול במצב חריג (exception handler)

כאשר מתעורר מצב חריג, ++C מפעילה את פונקציית הטיפול המתאימה למצב חריג זה. פונקציית הטיפול מוגדרת ב**מחלקת המצב החריג** (Exception class). לדוגמה, במחלקת המצב החריג הבאה, nuke_meltdown, מוגדרת פונקציית הטיפול הבאה:

```
class nuke_meltdown {
 public:
   nuke_meltdown(void) { cerr << "\a\a\aRun! Run! Run!""
                              << endl; }
};
```

כאשר התוכנית מעוררת את המצב החריג nuke_meltdown, ועוד לפני שהיא מחזירה את הבקרה לפקודה הבאה אחרי משפט try, ++C מבצעת את משפטי הפונקציה nuke_meltdown. התוכנית הבאה, **MELTDOWN.CPP**, מדגימה שימוש במצב החריג nuke_meltdown ובפונקציית הטיפול המתאימה לו. התוכנית משתמשת במשפט try המאפשר את איתורו ותפיסתו של המצב החריג. הפונקציה add_u232 תסיים בהצלחה, אם הארגומנט שהיא מקבלת קטן מ-255. אולם, אם הארגומנט גדול או שווה ל-255 הפונקציה nuke_meltdown תעורר מצב חריג. הנה לפניך פירוט התוכנית.

שים ♥! תוכנית זו פועלת בגירסה 5 ומעלה של ++Visual C או בגירסה Borland C++ 4.5 ומעלה בלבד.

```
#include <iostream.h>

class nuke_meltdown {
 public:
   nuke_meltdown(void) { cerr << "\a\a\aRun! Run! Run!""
                              << endl; }
};

void add_u232(int amount)
  {
    if (amount < 255)
      cout << "Amount of u232 is OK" << endl;
     else
       throw nuke_meltdown();
   }
void main (void)
  {
    try {
      add_u232(255);
     }

    catch (nuke_meltdown) {
       cerr << "Run Faster" << endl;
      }
    }
```

לאחר הידור והרצת התוכנית, יוצג על המסך הפלט הבא:

```
C:\> MELTDOWN < Enter>
Run! Run! Run!
Run Faster
```

אם תעיין בתשומת לב בתוכנית שיצרה את ההודעות על המסך, תוכל לעקוב אחר מהלך הטיפול במצב החריג ותפיסת התקלה. שורת הפלט הראשונה היא תוצאה של פונקציית הטיפול במצב החריג (הפונקציה nuke_meltdown). שורת הפלט השנייה היא תוצאת משפט catch (התופס את המצב החריג).

הגדרת פונקציה לטיפול במצבים חריגים

התעוררות מצב חריג (משפט throw) מלווה אוטומטית בריצת פונקציית הטיפול המתאימה לו. כדי להגדיר פונקציה לטיפול במצב חריג, צריך להגדיר במחלקת המצב החריג פונקציה, שהיא בעלת שם זהה לשם המחלקה (בדומה לפונקציית בנייה של מחלקה). פונקציית הטיפול צריכה לנסות ולתקן את הבעייה, כדי שהתוכנית תוכל לחזור על הפעולה שגרמה למצב החריג. לאחר סיום פונקציית הטיפול הזו, ריצת התוכנית ממשיכה מהמשפט העוקב למשפט try.

שימוש במשתני מחלקת מצב חריג

בדוגמאות שהוצגו עד כה, השתמשנו במשפט catch כדי לזהות את סוג המצב החריג שנוצר (אם בכלל), ולהגיב בהתאם. ככלל, ככל שכמות המידע שיש בידינו על המצב החריג, גדולה יותר, כך נוכל להגיב בצורה טובה יותר. לדוגמה, כשמתעורר המצב החריג file_open_error היינו רוצים, שהתוכנית תדע את שמו של הקובץ שהניסיון לפתוח אותו גרם לשגיאה. באותה מידה, כשמתעוררים מצבים חריגים file_read_error ו-file_write_error, היינו רוצים שהתוכנית תדע את המקום המדויק בזיכרון (Byte location) שבו קרתה השגיאה. אחסון מידע אודות מצב חריג מתבצע על ידי הוספת משתני מחלקה במחלקת המצב החריג. כעת, כאשר התוכנית מעוררת מצב חריג, היא תעביר את המידע הדרוש בצורת ארגומנטים אל פונקציית הטיפול שלו :

```
throw file_open_error(source);
throw file_read_error(344);
```

בדומה לפונקציית בנייה, גם פונקציית הטיפול במצב החריג תכלול משפטים אשר יציבו את ערכי הפרמטרים הפורמליים (הארגומנטים) לתוך המשתנים המתאימים במחלקה. לדוגמה, המשפטים הבאים משנים את המצב החריג file_open_error, כדי שיציב את שם הקובץ שגרם למצב החריג לתוך משתנה המחלקה filename :

```
class file_open_error {
 public:
   file_open_error(char *filename) {
     strcpy(file_open_error::filename,filename); }
   char filename[255];
};
```

דוגמה : excpdt.cpp

בדוגמה זו בלוק ה- catch מקבל כפרמטר מידע על מהות השגיאה.

הערה *: תוכנית זו פועלת ב- Visual C++ 5.0 ומעלה.*

```
#include <iostream.h>

float calc(float a, float b)
 {
  if (b == 0)
   throw "Error - division by zero in function - calc()";
  else
   return (a/b);
 }
```

```
void main(void)
 {
  try
  {
    cout << "7/3 = " << calc(7, 3) << endl;
    cout << "4/0 = " << calc(4, 0) << endl;
  }
  catch(char *msg)
  {
    cout << msg << endl;
  }
 }
```

טיפול במצבים חריגים בלתי-צפויים

בשעור 11, למדנו שמהדרי ++C מספקים ספריות זמן-ריצה, המכילות פונקציות בהן ניתן להשתמש ישירות בתוכנית. פונקציות אלו מסוגלות לעורר מצבים חריגים למיניהם. לפיכך, במידה והתוכנית שכתבת קוראת לפונקציית ספריה מסוימת, היא צריכה גם **לתפוס** את המצבים החריגים העלולים להתעורר כתוצאה מהפעלת הפונקציה. מצב חריג שנוצר ואינו זוכה לפונקציית טיפול מתאימה, גורם להפעלת פונקציית הטיפול של ברירת המחדל (Default exception handler). פונקציה זו מסופקת כברירת מחדל על ידי ++C. ברוב המקרים, פונקציית טיפול זו, תגרום לסיום ריצת התוכנית. התוכנית שלהלן – **UNCAUGHT.CPP**, מדגימה זאת.

```
#include <iostream.h>

class some_exception { };

void main(void)
 {
   cout << "About to throw exception" << endl;
   throw some_exception();
   cout << "Exception thrown" << endl;
 }
```

כאשר מתעורר המצב החריג some_exception, מופעלת פונקציית הטיפול של ברירת המחדל. פונקציה זו דואגת לסיום התוכנית בצורה "מסודרת", ככל שניתן. לפיכך, המשפט האחרון של התוכנית אינו מתבצע אף פעם. במקום להשתמש בפונקציית ברירת המחדל המסופקת עם מהדר ++C, ניתן להשתמש בפונקציית ברירת מחדל לטיפול במצבים חריגים, אשר מוגדרת על ידי התוכנית עצמה. כדי לעשות כך, צריך להשתמש בפונקציית הספריה set_unexpected המוגדרת בקובץ הכותר except.h.

פונקציה לטיפול במצבים חריגים בלתי-צפויים

פונקציה בשם **unexpected()** תופעל במידה והפונקציה תשלח התראה מטיפוס שונה ממה שהוגדר עבורה. ברירת המחדל שלה: הפסקת ריצת התוכנית.

אפשר להגדיר פונקציית unexpected() פרטית. ערכה המוחזר צריך להיות void והיא צריכה להיות פונקציה שלא מקבלת פרמטרים.

ואז ב- main() יש לכתוב את התחביר הבא:

```
set_unexpected(שם הפונקציה הפרטית שהגדרנו);
```

דוגמה: unxfun.cpp

הערות: *- תוכנית זו פועלת ב- Visual C++ 5.0 ומעלה.*

- מכיון שאין עדיין סטנדרטים לנושא זה, Visual C++ מתעלמת מאפשרות הגבלת throw().

```
#include <iostream.h>
#include <eh.h>
#include <stdlib.h>

void privateUnexpected(void)
 {
  cout<<"exception unexpected"<<endl;
  exit(0);
 }

float findMin(float a, float b) throw(float) // warning
 {
  if (a < 0)
    throw "a < 0";
  if (b < 0)
    throw "b < 0";
  if (a > b)
    return b;
  else
    return a;
 }

void main()
 {
  set_unexpected(privateUnexpected);
  float f;
```

```
  try
    {
      f = findMin(9.4f, -2.7f);
    }
  catch(char *msg)
    {
      cout << msg << endl;
    }
}
```

קביעת המצבים החריגים אשר מותר לפונקציה לעורר

כזכור, משפט הצהרה של פונקציה (function declatarion, function prototype) מגדיר את טיפוסי הפרמטרים וטיפוס הערך המוחזר של הפונקציה. בנוסף לכך, ניתן להשתמש בהצהרה של הפונקציה כדי לציין וגם לקבוע, איזה מצבים חריגים מותר לפונקציה לעורר. לדוגמה, ההצהרה הבאה מציינת למהדר כי לפונקציה power_plant מותר לעורר את מצב החריגים melt_down ו-..radiation_leak:

```
void power_plant(long power_needed) throw (melt_down,
                                    radiation_leak);
```

סגנון כתיבה כזה כדאי ביותר (מבחינת הנדסת תוכנה לפחות). כאשר הצהרת פונקציה כוללת את ציון המצבים החריגים שהיא עשויה לעורר, תוכניתנים אחרים מסוגלים לדעת בקלות איזה מצבים חריגים עליהם לבחון ולתפוס, כאשר הם משתמשים בפונקציה זו.

מצבים חריגים של מחלקות

C++ מאפשרת להגדיר מצבים חריגים השייכים בלעדית למחלקה מסוימת. כדי ליצור מצב חריג השייך במיוחד למחלקה מסוימת, צריך לכלול אותו כאחד מאיברי המחלקה בחלק הציבורי (Public) של המחלקה. לדוגמה, הגדרת המחלקה string שלהלן, כוללת שני מצבים חריגים, string_empty ו- string_overflow:

```
class string {
 public:
  string(char *str);
  void file_string(*str);
  void show_string(*str);
  int string_lenght(void);
  class string_empty { };
  class string_overflow {};
```

```
 private:
  int lenght;
  char string[255];
};
```

כפי שיכולת לראות בדוגמה, המחלקה מגדירה את המצבים החריגים string_empty ו- string_overflow. בתוכנית אפשר לאתר ולתפוס מצב חריג על ידי שימוש באופרטור **טווח ההכרה הגלובלי** ובשם המחלקה. למשל:

```
try {
 some_string.fill_string(some_long_string);
 };

catch (string::string_overflow) {
  cerr << "String lenght exceeded-characters truncated" << endl;
  }
```

פולימורפיזם ומנגנון ה- Exception Handling

פולימורפיזם הינה תכונה רבת עוצמה המייעלת מאוד את מנגנון הטיפול בשגיאות. בתוכנית הבאה ננצל תכונה זו כך שהאובייקט שנקלוט יהיה תמיד מטיפוס מחלקת הבסיס, ובתוך בלוק ה- catch תופעל הפונקציה ()print, שתהיה זו השייכת לאובייקט שנשלח בפועל על ידי throw ולכן ההודעה שתודפס תהיה תמיד בעלת הפירוט הרב ביותר. בכל איטרציה של לולאת ה-for נשלח כהתראה אובייקט מטיפוס של מחלקה שונה ולכן בכל פעם מופעלת פונקציית ()print המתאימה לאובייקט שנשלח.

דוגמה: unxpoylm.cpp

הערה *: התוכנית פועלת ב- Visual C++ 5.0 ומעלה.*

```
#include <iostream.h>
#include <string.h>

class errbase
   {
   public:
     virtual void print() {cout<<"errbase"<<endl;}
   };
class errname: public errbase
   {
   public:
     void print(){cout<<"errname - long name"<<endl;}
   };
```

```
class errquant: public errbase
  {
   public:
     void print(){cout<<"errquant"<<endl;}
  };

class errquant1: public errquant
  {
   public:
     void print(){cout<<"errquant1 > 5000"<<endl;}
  };

class errquant2: public errquant
  {
  public:
     void print(){cout<<"errquant2 < 0"<<endl;}
  };

class errprice: public errbase
  {
  public:
     void print(){cout<<"errprice"<<endl;}
  };

class errprice1: public errprice
  {
   public:
     void print(){cout<<"errprice1 > 500"<<endl;}
  };

class errprice2: public errprice
  {
   public:
     void print(){cout<<"errprice2 < 0"<<endl;}
  };

class product
  {
  private:
   char name[15];
   long quant;
   double price;
  public:
   void get(char *nm, long qu, double pr)
     {
      if (strlen(nm) > 14)
        throw errname();
```

```
      else
        strcpy (name, nm);
      if (qu > 50000)
        throw errquant1();
      else
        if (qu < 0)
          throw errquant2();
        else
          quant = qu;
      if (pr > 500)
        throw errprice();
      else
        if (pr < 0)
          throw errprice2();
        else
          price = pr;
     }
   void show()
     {
      cout<<"name = "<<name<<" quant= " <<quant<<" price= "
                                              <<price<<endl;
     }
  };

void main(void)
   {
   product p[6];
   char *nm[6] = {"cellular", "computer","screen","sea food",
                              "name_name_and_more_name","book"};
   long qu[6] = {72100L, 37L, 90L, -3L, 230L, 69L};
   double pr[6] = {46.25f, -7.80f, 9.25f, 3.12f, 887.39f, 1.15f};

   for (int i = 0; i < 6; i++)
     {
      try
        {
         cout<<endl;
         cout<<"input products"<<endl;
         cout<<nm[i]<<"   "<<qu[i]<<"   "<<pr[i]<<endl;
         p[i].get(nm[i], qu[i], pr[i]);
         p[i].show();
        }
      catch(errbase& msg)
        {
         msg.print();
        }
     }
   }
```

מצבים חריגים בתוך הבנאי

לבנאי אין כל ערך מוחזר. לכן, האמצעי היחיד בו הוא יכול להשתמש כדי להודיע לפונקציה, שממנה הוא הופעל, על כך שהתרחשה בו חריגה – הוא שליחת התראה. במקרה של שליחת התראה מתוך הבנאי (Throw) בניית האובייקט עדיין לא הושלמה ולכן המפרק לא יופעל. לכן, במידה והוקצה זיכרון צריך לשחרר אותו לפני ששולחים התראה. בהגדרת בנאי השייך למחלקה הנגזרת, במידה והתרחשה חריגה באתחול נתוני המחלקה, צריך לזכור לשחרר רק משאבים שהוקצו בתוך הבנאי ולא צריך לדאוג למשאבים שהוקצו על ידי הבנאי של מחלקת הבסיס, לכך דאגנו במימוש הבנאי של מחלקת הבסיס.

דוגמה: excconst.cpp

הערה *: התוכנית פועלת ב- Visual C++ 5.0 ומעלה בלבד.*

```
#include <iostream.h>

class object
   {
    private:
     float *fp;
    public:
     object (float f);
     ~object();
   };

object::~object()
   {
    cout<<"destruct object:" << *fp << endl;
    delete fp;
   }

object::object(float f)
   {
    cout<<"construct object:"<< f << endl;
    fp = new float(f);
    if (f < 0)
     {
       delete fp;
       throw "throw f < 0 constructor is not working";
     }
   }
```

```
void main(void)
  {
   try
    {
      object o1(7.4f) , o2(-9.1f);
    }
   catch(char *msg)
    {
      cout << msg << endl;
    }
  }
```

סיכום

מנגנון המצבים החריגים נועד לפשט ולשפר את היכולת של תוכניות ++C לגלות ולטפל בשגיאות זמן ריצה. כדי לעורר ולתפוס מצבים חריגים, משתמשים לשם כך במשפטים try, catch ו- throw. הכרת נושאים אלה תורמת לשיפור רמת התכנות.

לפני שנסיים, ראוי שנבדוק שוב את ידיעותינו בנושא exceptions :

- ✓ מצב חריג הינו שגיאה בלתי צפויה המתרחשת בזמן ריצת התוכנית.
- ✓ תוכניותיך צריכות לעורר, לתפוס ולטפל במצבים חריגים.
- ✓ משפט try מאפשר לתפוס מצבים חריגים.
- ✓ משפטי catch (אחד או יותר) צריכים להופיע מיידית אחרי משפט try, כדי לתפוס את המצב החריג שהתעורר (אם בכלל), וגם לזהות אותו.
- ✓ כדי לעורר מצב חריג, משתמשים במשפט throw.
- ✓ כאשר התוכנית מזהה מצב חריג, היא גורמת אוטומטית להפעלת פונקציית טיפול (Exception handler) המתאימה לו.
- ✓ הצהרת פונקציה יכולה לכלול את המצבים החריגים שמותר לה לעורר.
- ✓ פונקציות ספריה (Run-time library functions) עשויות לעורר מצבים חריגים.
- ✓ מצב חריג שנוצר, ואשר אינו מזהה פונקציית טיפול מתאימה, עלול לגרום להפעלת פונקציית הטיפול של ++C, שנקבעה כברירת מחדל (C++ Default exception handler).
- ✓ קובץ הכותר except.h כולל הצהרות פונקציות, בהן ניתן להשתמש כדי להגדיר פונקציית ברירת מחדל עצמאית לטיפול במצבים חריגים.

תרגילים

שים ♥ ! **לפתרון תרגילים אלה השתמשו בגירסה 5 ומעלה של ++Visual C או בגירסה ++Borland C 4.5 ומעלה בלבד.**

1. כיתבו פונקציה divide המקבלת שני מספרי נקודה צפה ומחזירה את תוצאת החלוקה של המספר הראשון במספר השני. הפונקציה מרימה אות חריג (exception) במקרה של חלוקה באפס. יש לתפוס את ה- exception בשגרת main.
2. כיתבו תוכנית המרימה אות חריג exception במקרה של חריגה בהקצאת זיכרון של אופרטור new.
3. כיתבו מחלקה array, המספקת שירותי גישה לאיברים במערך, תוך כדי בדיקת חריגות של גישות מעבר לגודל המערך. המחלקה תופעל על ידי תבנית (template), ותבצע העמסה של האופרטור [] לגישה לאיברים. יש לזרוק מצב חריג (exception), במקרה של חריגה מגבולות המערך.

פרק 40

ספריית תבניות סטנדרטיות STL

פרק זה נלקח מתוך הספר "**C++ ו-OOP למתכנת המקצועי**" שכתב **שמעון כהן**.

הפרק ניתן כתוספת למתקדמים ולא כחלק מהלימוד בספר. לא צירפנו את התוכניות בתקליטור ואנו מציעים לא לנסות להריץ אותן כי הן מוצגות באופן חלקי כאן.

יש לראות בפרק זה כמבוא לנושא זה.

בפרק זה נלמד על **ספריית התבניות הסטנדרטיות** - **STL** (**S**tandard **T**emplate **L**ibrary) של C++, ונסביר את עקרונות התכנון של הספריה שבה נעשה שימוש נרחב בתבניות C++. למרות שהספריה סטנדרטית והתקבלה על ידי המועצה של **ANSI**, מרבית המהדרים עדיין אינם תומכים בה. בעת כתיבת ספר זה תומכים בספריה המהדרים של בורלנד, המהדר של GNU (g++), והמהדר Visual C++ של מיקרוסופט. חברת Sun הכריזה על מהדר שיתמוך בה גם הוא. הדוגמאות בפרק זה הורצו בעזרת המהדרים של **בורלנד**.

עקרונות הספריה STL

הספריה הסטנדרטית STL מורכבת ברובה ממחלקות תבנית ופונקציות תבנית. יש בה שימוש נרחב בתבניות ולכן נדרש לפחות ידע בסיסי בנושא, אם לא למעלה מזה.

הספריה אינה מתבססת על ירושה, אלא על **תכנות גנרי** (Generic programming). תכנות גנרי משתמש בפונקציות המקבלות איטרטורים לתחום מסוים שעליהם הן עובדות. האיטרטורים מתנהגים כמו מצביעים, ולכן פונקציות גנריות טובות גם למצביעים וגם למחלקות. על עקרונות אלו מבוססת הספריה הסטנדרטית של C++.

הספריה עצמה מחולקת לשלושה מרכיבים עיקריים, כשיש מספר מרכיבים משניים. מרכיביה העיקריים הם:

- **מכולות** (Containers, או **אוספים**) - אלו הם אובייקטים שמכילים אובייקטים אחרים, כמו למשל רשימה או מערך.
- **איטרטורים** - אובייקטים מסוג זה מאפשרים איטרציות על אוספים המכילים אובייקטים אחרים.
- **אלגוריתמים** - אלה הם פונקציות תבנית, שאינן קשורות במחלקה מסוימת באופן מיוחד. פונקציות אלו מבצעות אלגוריתם, כמו find, המוצא אובייקט באוסף כלשהו.

STL, אם כן, היא ספריה של אוספי אובייקטים שהם למעשה מחלקות שנבנו בצורה גנרית, ולכן אפשר להשתמש בהן כמעט בכל יישום. בהיסטוריה של C++ כבר היו מספר לא קטן של ספריות כאלו, כמו למשל ספריית האובייקטים NIH, ספריית האובייקטים של בורלנד וספריית האובייקטים של Rogue Waves.

החיסרון של כל הספריות שהזכרנו בכך שלא היו סטנדרטיות. הן פעלו בסביבה מסוימת, אך כשהעבירו את התוכנה שהשתמשה בהן לסביבה אחרת, היה צורך להעביר גם את הספריה. במקרים רבים היה הדבר קשה, ולעיתים בלתי אפשרי. ספריית STL תצורף לכל מהדר עם ממשק ANSI, ולכן, תוכנה שתיכתב בעזרת STL בסביבה מסוימת תפעל גם בסביבה אחרת.

חיסרון אחר של הספריות הקודמות הוא ה**אלגוריתמים**. הספריות הקודמות לא תמכו באלגוריתמים כגון sort, או unique. הן סיפקו מבני נתונים שאיפשרו להכניס, להוציא ולמצוא אובייקטים, וגם סיפקו איטרטורים.

הגישה בספריית האובייקטים הסטנדרטית של C++ **שונה** מהגישה המקובלת בתכנות מוכוון אובייקטים. לפי גישה זו, יש מחלקה מסוימת ופעולות ששייכות למחלקה, אך אין פונקציות חיצוניות שפועלות עליה. גישה זו אכן שומרת על עיקרון הסתרת המידע של המחלקה, אך אינה מאפשרת לנצל תכונות משותפות בין משתנים בסיסיים של השפה לבין מחלקות המייצגות אוספים של אובייקטים.

גישה זו מתאימה **לשפת תכנות מוכוונות אובייקטים טהורה** (pure object oriented language), כמו Smalltalk. ב-Smalltalk כל משתנה, או מצביע לאובייקט, שנגזר מאובייקט בסיסי נקרא Object. לכן, כל פונקציה שייכת למחלקה Object או מחלקה אחרת היורשת ממנה. ממילא, כל פונקציה שפועלת על אובייקטים מסוג Object פועלת על אובייקט היורש ממנה. לכן, אין מקום וצורך בפונקציות גנריות. לדבר זה יש יתרונות וחסרונות, והדעות בנושא זה חלוקות. ספר זה עוסק ב-C++ ולא בפילוסופיה, ולכן לא נעסוק בוויכוח זה.

שפת C++ מספקת משתנים בסיסיים ופונקציות, בנוסף לתמיכה בתכנות מוכוון אובייקטים, ולכן היא נקראת C++ **שפה משולבת** (Hybrid). כשאנו כותבים פונקציית מיון ב-C++ עבור מערך של אובייקטים כמו String, נרצה להשתמש בפונקציה זו גם עבור מערך של מצביעים לתווים. איננו רוצים להעתיק את מערך המצביעים לתווים למערך של אובייקטים ולבצע את המיון רק לאחר מכן. לכן, אם נכתוב פונקציית מיון עבור מחלקה המייצגת מערך של אובייקטים, היא לא תהיה טובה עבור מערך של משתנים בסיסיים של השפה.

תכנות גנרי הוא המונח העומד מאחורי הספריה הסטנדרטית, והוא הרעיון המרכזי בספריה זו. בתכנות גנרי אנו מנסים להתעלם מסוג האובייקטים, או המשתנים עליהם פועל האלגוריתם, ולהתרכז בפעולה הבסיסית של האלגוריתם עצמו. למשל, אלגוריתם מיון צריך רק לקבל יחס סדר בין האובייקטים ותחום המתאר את קבוצת האובייקטים שצריך למיין. סיבות אלו מראות מדוע ספריות האובייקטים לא הצליחו ב-++C כמו בשפות תכנות אחרות. **STL** פותרת בעיות אלו ומספקת מבני נתונים רבים ומורכבים ביעילות רבה, וגם אלגוריתמים הפועלים על מבני נתונים אלה ועל מערכים בסיסיים של השפה.

הקוד שפותח ב-STL יעיל מאוד, לכן אין כל צורך לשכתב את הספריה הבסיסית כדי להגיע לביצועים גבוהים יותר, פרט לבעיה אחת. הבעיה בקוד זה היא שכולו מבוסס על תבניות, ולכן כמות הקוד שיוצר המהדר היא רבה כפי שנראה בהמשך.

ספריית אובייקטים המספקת מיגוון רב של אובייקטים מקלה באופן משמעותי על כתיבת התוכנה, חוסכת זמן וטעויות וגם יעילה מאוד. ספריה כזו הופכת את התכנות לפעולה נעימה, וחוסכת את הצורך לשוב ולדאוג למחלקות שכבר כתבנו.

מכולות ב-STL

ב-STL יש מספר **מכולות** (Containers) השונות זו מזו ביכולות שלהם, אשר מתאימות למצבים שונים. הממשק לאובייקטים השייכים למחלקות אלו דומה בדרך כלל לממשקים שאנו מכירים, ולכן הדבר מאפשר פיתוח אלגוריתמים שעובדים עבור מספר רב של מכולות ואינם קשורים למחלקה יחידה. במילים אחרות, האלגוריתמים אינם פונקציות של מחלקה. המחלקות המייצגות מכולות, כוללות פונקציות הכנסה והוצאה, התלויות במבנה הנתונים של המחלקה.

כל המכולות מכילות אובייקטים, ולא מצביעים לאובייקטים. כלומר, כשמכניסים אובייקט למכולה, המכולה יוצרת תחילה **העתק** של האובייקט, ואת ההעתק הזה היא מצרפת לאוסף האובייקטים שלה. מסיבה זו אין שיתוף בין אובייקטים במספר מכולות. המתכנת יכול ליצור אוסף של מצביעים ואז ליצור שיתוף בין אובייקטים במכולות שונות. במצב כזה אחראי המתכנת להקצאה ושחרור של אובייקטים. נניח, למשל, שקיימת מכולה כלשהי:

```
container<String*> c1, c2;
String *sp = new String ("ptr");
c1.push_back(sp);
c1.push_back(sp);
```

הכנסנו מצביעים למכולות ולכן המתכנת גם חייב לבטל את האובייקטים האלה. כשמוכנסים אובייקטים (ולא מצביעים), אחראי האוסף לשחרור והקצאת זיכרון עבור הצמתים שבאוסף, וגם עבור האובייקטים שצורפו לאוסף. המכולות מאפשרות למשתמש בהן לשנות את הדרך שבה מוקצה עבורן זיכרון. כך יכול המשתמש לקבוע את אופן הקצאת הזיכרון. לפי מודל התוכנית אפשר לקבוע את סוג המצביעים ל-near או far במונחים של מחשב אישי. הדבר מאפשר גם לקבוע זיכרון שיהיה זמין גם מעבר לתקופת החיים של התוכנית, כלומר **Persistent**.

האוספים הנתמכים ב-STL הם:

- ❖ **vector** - אוסף הדומה לווקטור. האובייקטים באוסף כזה אגורים בצורה סדרתית בזיכרון רציף. הגישה לאובייקטים באוסף כזה היא אקראית בעזרת אופרטור [].
- ❖ **list** - רשימה מעגלית כפולה. הגישה לאובייקטים באוסף זה היא סדרתית, קדימה או אחורה.
- ❖ **deque** - תור כפול שמאפשר להכניס אובייקטים משני הצדדים (תחילת התור וסופו). התור הכפול גם מאפשר להוציא אובייקטים משני צדדיו.
- ❖ **set** - אוסף סדור (קבוצה) של אובייקטים. לאובייקטים המוכנסים לאוסף זה יש יחס של סדר. הדבר אינו דומה להגדרה המתמטית של אוסף (ההגדרה המתמטית של קבוצה היא: אוסף שאינו סדור של אובייקטים, ללא חזרות). באוסף זה אסורות החזרות של אובייקטים. התכונה החזקה של אוסף זה היא מהירות גבוהה של חיפוש אובייקטים.
- ❖ **multiset** - אוסף הדומה לאוסף הקודם. אוסף סדור של אובייקטים, אך החזרות מותרות. כלומר, מותר להכניס את אותו אובייקט מספר פעמים.
- ❖ **map** - מפה, לעיתים נקרא מבנה נתונים זה בעגה המקצועית - **מילון**. זהו אוסף של זוגות אובייקטים, שכל אחד מהם מורכב ממפתח וערך. המפה היא אוסף סדור לפי המפתחות. באוסף מסוג זה ייתכן רק מפתח יחיד מערך מסוים.
- ❖ **multimap** - אוסף סדור של זוגות אובייקטים, כשכל זוג מורכב ממפתח ומערך. במקרה זה מותר מספר זוגות עם מפתח זהה.

כל המכולות, או האוספים, מאפשרים להכניס כל אובייקט לתוכם, כולל סוגים בסיסיים של השפה, כמו שלמים או float.

איטרטורים

איטרטורים (Iterators) הם אובייקטים שמאפשרים סריקה של אוספים, כלומר – של מכולות. לכל אחד מאוספים אלה יש הגדרה של איטרטור. איטרטור מסוג מסוים מתאים אך ורק לאוסף מסוים. דבר זה מאפשר להפעיל פונקציות בצורה גנרית, ללא כל צורך בידיעת סוג האוסף.

STL רואה את האיטרטורים כסוג מצביע מסוים. כשרוצים לקדם מצביע, מפעילים עליו אופרטור "++". כשרוצים להחזיר את המצביע לאחור מפעילים עליו את האופרטור "--". האופרטור "מצביע" לאובייקט נוכחי שנמצא באוסף. לכל איטרטור יש מידע המציין את האובייקט הנוכחי של האיטרציה.

כשרוצים להשתמש בתכולת מצביע משתמשים באופרטור "*". בדומה, כשרוצים לקבל את האובייקט הנוכחי של האיטרציה משתמשים באופרטור "*". הדבר מאפשר לפתח אלגוריתמים שונים המתאימים למצביעים וגם לאיטרטורים.

STL מגדירה את סוגי האיטרטורים הבאים:

- **Random access iterator** (**איטרטור לגישה אקראית**) - איטרטור מסוג זה תומך באופרטורים "++", "--" ו-"[]" והוא בעל כל התכונות של מצביעים בסיסיים בשפה. האופרטור "++" מאפשר להתקדם באוסף שאליו מצביע האיטרטור. האופרטור "--" מאפשר לנוע לאחור באוסף שאליו מצביע האיטרטור. האופרטור "[]" מאפשר גישה לכל מקום באוסף שאליו מצביע האיטרטור. הגישה במקרה זה היא על פי אינדקס.
- **Bidirectional direction iterator** (**איטרטור דו-כיווני**) - איטרטור מסוג זה מאפשר לנוע לשני הכיוונים באוסף שאליו הוא מצביע. התנועה עם איטרטור זה מתאפשרת על ידי האופרטורים "++" ו- "--". איטרטור זה אינו מאפשר גישה אקראית בעזרת אופרטור "[]". איטרטור זה נחשב לבעל יכולות נמוכות יותר מהקודם.
- **Forward iterator** (**איטרטור מתקדם**) - איטרטור מסוג זה מאפשר לנוע קדימה בלבד באוסף שאליו הוא מצביע. הוא עושה זאת בעזרת האופרטור "++".
- **Backward iterator** (**איטרטור נסוג**) - איטרטור מסוג זה מאפשר תנועה לאחור בלבד באוסף אליו הוא מצביע. הוא עושה זאת בעזרת האופרטור "--".
- **Input iterator** (**איטרטור קלט**) - איטרטור זה מאפשר השמה בלבד של אובייקטים ממנו אל אובייקט כלשהו. בדרך כלל, הוא קשור לקובץ ומאפשר לחלץ ממנו נתונים.
- **Output iterators** (**איטרטור פלט**) - איטרטור זה מאפשר השמה של אובייקטים לתוכו. בדרך כלל, האיטרטור קשור לקובץ ומשמש לפלט.

לכל אוסף **container** מוגדר האיטרטור בצורה הבאה:

```
container::iterator i;
```

דבר זה מאפשר לאלגוריתמים שונים להשתמש באיטרטור של מכולה נתונה, מבלי לדעת את סוג המכולה או את סוג האיטרטור.

שימוש באלגוריתמים ואיטרטורים

האלגוריתמים השונים יכולים לפעול גם על מצביעים בסיסיים של השפה. במקרים כאלה יש להבדיל בין סוגי האיטרטורים כדי לממש את האלגוריתמים באופן יעיל. כדי להבין את משמעות הדבר נבחן דוגמה. נניח, שיש לנו פונקציה המקדמת איטרטור, או מצביע, במספר נתון n.

```
template <class Iterator>
void advance(Iterator &iter, int n)
 {
   iter += n;
 }
```

הפונקציה הגנרית הזו יכולה לעבוד עבור **איטרטורים** מסוג **מצביעים**, או עם **איטרטורים** מסוג **גישה אקראית**. עבור איטרטורים דו-כיוונים אין פעולה "=+", ולכן קטע קוד זה אינו עובר הידור. אפשר לשנות את הפונקציה הזו בצורה הבאה:

```
template <Iterator>
void advance(Iterator &iter, int n)
 {
    if (n > 0)
       while (n-- > 0)
             ++iter;
    else
       while (n++ > 0)
             --iter;
 }
```

פונקציה זו מתאימה לפעולה עם **איטרטורים מסוג גישה אקראית** (random access iterator), או **איטרטורים דו-כיוונים**. הבעיה נוצרת כשמפעילים פונקציה זו על איטרטור דו-כיווני. במקרה כזה מספיק המשפט הזה:

```
iter += n;
```

במקרים כאלה יעילות הפונקציה גרועה במיוחד, כי נדרשות n פעולות במקום פעולה אחת. כדי לפתור בעיה זו מוגדר המושג **תווית** (**Tag**) עבור איטרטורים (ראה להלן).

תווית של איטרטור

כפי שניתן לראות, קיימת בעיית הבחנה בין הפונקציות השונות. אם נכתוב פונקציה גנרית שתקדם איטרטור, היא תהיה בזבזנית מאוד עבור איטרטורים לגישה אקראית. פתרון אחד הוא להגדיר היררכיה של מחלקות איטרטורים. לדוגמה:

```
template <class Type>
class forward_iterator {
public:
    forward_iterator() {}
    forward_iterator &operator++()
    { return *this; }
    // ...
};
template <class Type>
class random_access_iterator {
public:
    random_access_iterator() {}
    random_access_iterator &operator+=(int n);
    ...
};
```

כעת יכולנו לגזור את האיטרטורים שלנו מהאיטרטורים הבסיסיים, למשל:

```
template <class Type>
class deque {
...
public:
    class iterator : public random_access_iterator<Type> {
    ...
    iterator &operator+=(int n)
    { cur += n; return *this; }
    ...
};
```

עכשיו אפשר להשתמש ביכולת של C++ להגדיר גירסה מסוימת של הפונקציה הגנרית advance, באופן הבא:

```
template <class Type>
void advance(random_access_iterator<Type> &j, int n)
 {
    j += n;
 }
```

הבעיה המתגלה בפתרון זה היא, שהוא אינו יכול לפעול עבור מצביעים בסיסיים של השפה. הסיבה לכך היא שהמצביעים הבסיסיים אינם יורשים ממחלקה כלשהי, ובפרט אינם יורשים מהמחלקה random_access_iterator.

הפתרון שמציגה הספריה **STL** לבעיה, הוא שימוש ב**תוויות איטרטורים** (iterator tags). תחילה מגדירים את התוויות השונות של האיטרטורים:

```
struct input_iterator_tag {};
struct output_iterator_tag {};
struct forward_iterator_tag {};
struct bidirectional_iterator_tag {};
struct random_acess_iterator_tag {};
```

לאחר מכן מוגדרים האיטרטורים השונים והקטגוריות הבסיסיות שלהם. קטגוריות אלו מספקות מחלקות בסיסיות עבור איטרטורים המוגדרים במכולות השונות.

```
template <class T> struct input_iterator {};
struct output_iterator {};
template <class T> struct forward_iterator {};
template <class T> struct bidirectional_iterator {};
template <class T> struct random_access_iterator {};
```

לכל איטרטור מוגדרת ה**קטגוריה** אליה הוא שייך. אין זו מחלקה בסיסית שממנה יורשים האיטרטורים, אלא הקטגוריה בלבד. כזכור, אין יחס של ירושה למצביעים הבסיסיים של השפה.

```
template <class T>
inline input_iterator_tag
iterator_category(const input_iterator<T>&) {
    return input_iterator_tag();
}

inline output_iterator_tag iterator_category(const
output_iterator&) {
    return output_iterator_tag();
}

template <class T>
inline forward_iterator_tag
iterator_category(const forward_iterator<T>&) {
    return forward_iterator_tag();
}

template <class T>
inline bidirectional_iterator_tag
iterator_category(const bidirectional_iterator<T>&) {
    return bidirectional_iterator_tag();
}

template <class T>
inline random_access_iterator_tag
iterator_category(const random_access_iterator<T>&) {
    return random_access_iterator_tag();
}

template <class T>
inline random_access_iterator_tag iterator_category(const T*) {
    return random_access_iterator_tag();
}
```

הערה



בפונקציה ובמחלקות המקוריות יש פרמטר תבנית נוסף, Distance. לצורך פשטות ההסבר הרשיתי לעצמי לבטל אותו. הורדת הפרמטר אינה משנה את עקרון התוויות לאיטרטורים.

לאיטרטורים מסוג גישה אקראית (Random access iterator) יש שתי פונקציות מועמסות המספקות תוויות. הפונקציה הראשונה מספקת תווית איטרטור מסוג גישה אקראית עבור איטרטורים מסוג מחלקות, והפונקציה השנייה מספקת תווית מסוג של איטרטור גישה אקראית עבור איטרטורים שהם מצביעים בסיסיים.

כעת, באמצעות תוויות האיטרטורים אנו יכולים לכתוב את הפונקציה advance בצורה הבאה:

```
template <clas Iter>
inline void advance(Iter &j, int n)
{
    _advance(j, n,iterator_category(j));
}

template <class RandomAcessIterator>
inline void _advance(RandomAcessIterator &j,
                     int n, random_access_iterator_tag)
{
    j += n;
}

template <class BidirctionalItrator>
void _advance(BidirectionalIterator &j,
        int n, bidirectional_iterator_tag)
{
    if (n >= 0)
      while (n--)
            ++j;
    else
      while (n++)
            --j;
}
```

כלומר, הפונקציה הראשית שנקראת מפנה את הקריאה לפונקציה אחרת, שלה יש את אותם הארגומנטים, בתוספת תווית זיהוי של האיטרטור. מנגנון ההעמסה של C++ בוחר את הפונקציה המתאימה לפי סוג התווית! טכניקה זו מאפשרת הפעלה של פונקציות שונות בצורה **פולימורפית סטטית** (Static polymorphism). סוג הפונקציה נבחר על ידי המהדר לפי סוג הפרמטרים (התווית של האיטרטור) **בזמן הידור**, ולא בזמן ריצה.

אלגוריתמים

אחד הדברים המייחדים את STL הם ה**אלגוריתמים** של ספריה זו. בניגוד לספריות אחרות, מספקת STL מיגוון רחב של אלגוריתמים גנרים המתאימים ליישומים ושימושים רבים. אלגוריתמים רבים מתוכם טובים גם לאוספים בסיסיים של השפה, כמו **מערכים**.

אפשר לקחת את האלגוריתמים של STL ולהשתמש בהם עבור ספריות אחרות, שחסרות אלגוריתמים כאלה. למשל, ספריית האובייקטים של Rogue Waves החדשה מאפשרת שימוש באלגוריתמים של STL.

בדרך כלל, האלגוריתמים ב-STL פועלים על תחום נתון. העיקרון המנחה הוא פעולה בעזרת איטרטורים. האלגוריתם מקבל, בדרך כלל, איטרטור להתחלת התחום ואיטרטור לסוף התחום. אלגוריתם כזה, למשל, הוא find המוצא אובייקט נתון בתחום. האלגוריתם ממומש בצורה הבאה:

```
template <class Type, class Iterator>
Iterator find(const Type &key, Iterator first,
        Iterator last)
{
    while (first != last && *first != key)
       ++first;
    return first;
}
```

האלגוריתם הוא **פונקציית תבנית** בעלת שני פרמטרים; הראשון מייצג את סוג האובייקט והשני מייצג את סוג האיטרטור. הפונקציה מקבלת ייחוס לאובייקט שצריך למצוא, ותחום המאופיין על ידי האיטרטורים first ו-last. הפונקציה מחזירה איטרטור לאובייקט הזהה למפתח שנמסר לה, אם נמצא אובייקט כזה, אחרת היא מחזירה איטרטור הזהה לסוף התחום.

הפונקציה עשויה להחזיר מצביע לסוף האוסף, ולכן האובייקט האחרון בתחום אינו נחשב בתוך התחום. דבר זה מתאפשר הודות לתכונה של ++C, שכתובת אחת אחרי המערך היא כתובת חוקית לקריאה, אבל **לא** לכתיבה.

נוהל עבודה זה בעזרת אלגוריתמים הביא את המושג **תכנות גנרי** (Generic programming), שבו כותבים פונקציות **שאינן** מכירות את מבנה הנתונים עליהן הן פועלות. הפונקציות, שמממשות אלגוריתם, מקבלות איטרטורים לתחילת התחום ולסופו. פונקציה כזו מניחה שיש לאיטרטור אופרטורים (++) המאפשרים לקדם אותו לאלמנט הבא. כמו כן, מניחה פונקציה כזו שיש אופרטור (*) המקבל את האובייקט הנוכחי, כשהאופרטור מופעל על האיטרטור. על פי הנחות אלו יכולה הפונקציה לפעול גם על איטרטורים של מחלקות "מכולה" (או אוספים) וגם על מערכים בסיסיים של השפה.

דוגמה אחרת לאלגוריתם כזה, היא אלגוריתם המבצע **היפוך סדר** של אובייקטים במכולה. אם האובייקטים מסודרים בסדר עולה, מסדר אותם האלגוריתם בסדר יורד.

```
template <class Iterator>
void reverse(Iterator first, Iterator last)
{
    --last;
    while (first != last) {
       swap(*first, *last);
       if (++first == last) break;
       --last;
    }
}
```

כפי שראינו קודם לכן, האלגוריתם מקבל תחום שעליו הוא פועל, ובכל שלב של הלולאה הוא מחליף את האובייקטים בקצות התחום וגם מקטין את התחום. כמו במקרה הקודם, גם כאן יכול אלגוריתם כזה לפעול על מערכים בסיסיים של השפה, או על אוספים. כשהוא פועל על מערכים בסיסיים של השפה הוא מקבל מצביעים, וכשהוא פועל על אוספים הוא מקבל את האיטרטור של האוסף.

מתאמים (Adaptors)

מתאמים (Adaptors) הם אובייקטים שמקבלים אובייקט אחר, ומשתמשים בו כדי לממש פונקציונליות מסוימת. המתאמים אינם נחשבים כאחד מהמרכיבים העיקריים של הספריה, אבל הם מציגים רעיון תכנות שראוי לציינו כאן.

המתאם עצמו אינו מממש מבנה אלגוריתמים כלשהו, אלא משמש **כמעטפה** של מחלקה אחרת שמממשת את מבנה הנתונים והפונקציונליות הבסיסית. המעטפה מתאמת את הממשק של המחלקה הבסיסית לממשק מבוקש. למשל, אם רוצים לממש מחסנית, עומדות בפנינו מספר אפשרויות. אפשר לממש מחסנית בעזרת מערך, בעזרת רשימה או בעזרת תור כפול. לכן, המחלקה מחסנית יכולה להיות מתאם שמקבל סוג של מחלקה ומממש את הפונקציונליות הדרושה.

```
template <class Container, class T>
class stack {
    Container c;
public:
    stack() {}
    void push(const T &v)
    { c.push_front(v); }
    T pop()
    { return c.pop_front(); }
};
```

המחלקה stack היא **מעטפה** (Envelop) למכולה אחרת, ומשתמשת בה כדי לממש את הפונקציונליות של המחלקה. כל הפונקציות של המחלקה מבוצעות בעזרת המכולה הנתונה לה. כשמגדירים אובייקט ממחלקה זו, מספקים לו את המכולה ואת סוג האובייקטים שבמכולה (ראינו שאפשר להסתדר גם ללא סוג האובייקט). למשל:

```
stack<vector<int>, int> svi;
stack<list<int>, int>   sli;
```

למעשה, הגדרנו מחסנית המבוססת על וקטור, או רשימה. לפי סוג השימוש שלנו במחסנית ניתן לבחור את מבנה הנתונים המתאים יותר.

אובייקטי פונקציות (Function Objects)

אובייקטי פונקציות (Function objects) אינם נחשבים כאחד החלקים העיקריים של הספריה, אך ראוי להכיר מונח חשוב זה. אובייקט פונקציה מתנהג בצורה דומה לפונקציה. קיים אופרטור () שאפשר להפעילו כפונקציה. למשל:

```
struct Functor {
    int opreator()()
    { ... }
    ...
};
```

אובייקט זה מעמיס את אופרטור הפונקציה, ולכן אפשר להשתמש בו בצורה הבאה:

```
Functor f;
if (f()) { ... }
```

יכולת זו חשובה כשרוצים להעביר פונקציה, או אובייקט, לאלגוריתמים המשמשים להשוואה בין אובייקטים. האלגוריתם יכול להתייחס לאובייקט כפונקציה, ואז הדרך פתוחה בפני המשתמש להעביר לאלגוריתם פונקציה או אובייקט.

האלגוריתם qsort בספריה הסטנדרטית של C מקבל פונקציה המאפשרת לקבוע את הסדר הערכי (מספרי או אלפביתי) בין שני אובייקטים, באמצעות מיון המערך בזיכרון. באופן דומה אפשר לכתוב אלגוריתם המבוסס על פונקציית תבנית ב-C++ למיון מערכים, או מכולות. עלינו לספק לאלגוריתם אובייקט, שמאפשר לקבוע את הסדר בין אובייקטים שונים, או לחילופין, לתת לו פונקציה שמאפשרת זאת.

```
template <class Iterator, class Comp>
void sort(Iterator start, Iterator end, Comp cmp)
{
    //...
    if (cmp(*start, *end) < 0)
    {
       // start is less than end
       ...
    }
    ...
}
```

לפני השימוש באלגוריתם עלינו להגדיר את האובייקט, או הפונקציה, שמאפשרים ביצוע השוואה בין אובייקטים, ולמסור אותם לפונקציית המיון.

סיכום

בפרק זה הצגנו את **ספריית האובייקטים הסטנדרטית** של ++C. סקרנו את החלקים העיקריים שלה ואת רעיונות התכנון שהינחו את מתכנניה. רעיונות אלה לכשעצמם יכולים לשמש אותנו כשאנו מפתחים תוכנה לצרכינו.

הספריה הסטנדרטית של ++C מורכבת משלושה חלקים עיקריים: **מכולות** (Containers), **איטרטורים** **ואלגוריתמים**. בניגוד לגישה מוכוונת האובייקטים שבה יש אובייקטים בלבד, ספריה זו מציגה גישה חדשה שמתאימה ל-++C. בגישה זו יש אלגוריתמים נפרדים **שאינם** קשורים לאובייקטים. אלגוריתמים אלה פועלים על **תחומים** נתונים. האלגוריתמים משמשים כהפשטה בפני עצמה, כי אין הם מכירים את **מבנה הנתונים** שעליו הם פועלים. האלגוריתם דורש איטרטורים, או מצביעים למבנה נתונים זה, שיאפשרו לו לסרוק את מבנה הנתונים.

אלמנטים אחרים המרכיבים את הספריה כוללים **מתאמים** (Adaptors) **ואובייקטי פונקציות** (Function objects). המתאמים מאפשרים שימוש במחלקה אחרת והתאמתה למבנה נתונים מסוים. המתאם מאפשר גמישות למשתמש, בכך שהוא מאפשר לבחור את המחלקה המתאימה. לאובייקט המשמש כפונקציה יש אופרטור קריאה לפונקציה, ולכן אפשר להעבירו כפונקציה לאלגוריתם שצריך פונקציה המתארת את יחס הסדר בין אובייקטים שעליו האלגוריתם פועל.

מקורות

1. Musser, D. R. and A. A. Stepanov. "Algorithm oriented generic libraries", Software Practice and Experience, 24(7):623, July 1994.

2. Musser, D.R and A. Saini. STL Tutorial and Refernce Guide: C++ Programing with the Standard Template Library. Addison Wesly, Reading, MA, 1996.

פרק 41

יצירת פרויקט

פרק זה נלקח מתוך הספר "**המדריך השלם לשפת C - מהדורה 5**" שכתבו **משה ליכטמן** ו**עמית רש**. הפרק נכתב על ידי **אבי בוץ**.

הפרק ניתן כתוספת לספר. התוכניות המוזכרות בפרק זה לא נמצאות בתקליטור.

עד כה למדנו את טכניקת השימוש בפונקציות ספריה. למדנו שיש לכלול בתוכנית את הספריה הדרושה. למשל, נכתוב <iostream.h> #include לצורך שימוש בפונקציות הנמצאות בספריית קלט/פלט.

מהי פונקציית ספריה?

פונקציה היא תוכנית מחשב אשר כתובה בשפת C או בשפת Assembly, או שילוב של שתיהן. תוכנית מחשב זו נכתבה עבורנו במהדר C של בורלנד או מיקרוסופט (או מהדר אחר), בדרך בה אנו כותבים את תוכניותינו. כך נוצר קובץ C ובו מספר פונקציות בעלות מכנה משותף, כמו טיפול בקלט/פלט, למשל.

בקובץ התוכנית שכתבנו, אשר קרוי גם **קובץ מקור** (Source file), אנו כוללים את הפונקציה הראשית main. את הקובץ הזה אנו ממירים לספריה. לקובץ מתווסף קובץ נוסף באותו שם, אך בעל הסיומת h. למשל, stdio.c ו-stdio.h. **בקובץ הכותר** h (Header file) נמצאות כל ההכרזות של הפונקציות, בדיוק כמו בשיטת Top-Down. הכללת ספריה זו בתוכנית נועדה לשלב את ההכרזות בקובץ, כדי שנוכל להשתמש בפונקציות שבקובץ stdio.c.

למעשה, מבלי שנדע, עבדנו עד היום עם מספר קבצי מקור שכללו פונקציות שלא נכתבו על ידינו. כיצד, אם כך, נוכל לבנות קובץ פונקציות משלנו, שנוכל להשתמש בו ככל שנמצא לנכון, בלי לכתוב שוב את הפונקציה, או מבלי להעביר אותה מתוכנית לתוכנית?

יצירת תוכנית ממספר קבצי מקור

שפת C מאפשרת לפרוש את כתיבת התוכנית על פני מספר **קבצי מקור** (Source files), בעלי סיומת C. כל קובץ כזה יכול להכיל פונקציה אחת או יותר, המוכרות באופן גלובלי זו לזו. קובץ מקור אחד בלבד חייב להכיל את הפונקציה ()main, שממנה מתחילה ריצת התוכנית. לדוגמה, ייתכן המצב המוצג להלן. הקובץ file1.C מכיל:

```
main()
{
 ...
      func1();
      func2();
 ...
}
func1()
 {
 ...
      func2()
 ...
 }
func3()
 }
 ...
 ...
 }
```

הקובץ file2.C מכיל:

```
func2()
{
 ...
}
```

כלומר, למרות העובדה שהפונקציה ()func2 נמצאת בקובץ file2.C, ניתן לקרוא לה מתוך הפונקציות ()main ו-()func1, המוגדרות בקובץ file1.C.

לאחר תהליך ההידור של התוכניות שבקבצים file1.C ו-file2.C, נוצרים קבצי ביניים בעלי השמות file1.obj ו-file2.obj בהתאמה. תהליך ה**קישור** (Link) מחבר את שני קבצי הביניים הללו ואת קבצי ספריית המערכת לקובץ ריצה אחד, המקבל סיומת **EXE**. ניתן להריץ אותו על ידי הקלדת שמו, כאשר נמצאים בפיקוח מערכת DOS. למעשה, כאן אנו רואים באופן ברור יותר שהשתמשנו בשיטה זו באופן סמוי. כאשר כתבנו תוכניות C בקובץ מקור אחד, קראנו למעשה לפונקציות של ספריות המערכת מתוך התוכניות. כידוע, פונקציות אלו הוגדרו בקבצי מקור אחרים, ושולבו בשעת ביצוע הקישור.

סוג נוסף של קבצים הינם קבצי ספריה, בעלי סיומת LIB. בכל אחד מקבצים אלה משולבים כמה קבצי obj., והמקשר יודע לשלוף מתוכם רק את הקטעים הנחוצים לתוכנית. בקבצים אלה לא נעסוק כאן, אולם חשוב לדעת שקבצי ספריות המערכת של Borland C למשל, הינם קבצים מהסוג הזה.

מהו פרויקט?

ב-Borland C מקובל לכנות בשם "פרויקט" תוכנית הפרושה על פני מספר קבצי מקור. התהליך ליצירת קובץ exe על ידי הידור וקישור נקרא **עשיית פרויקט** (Project Make).

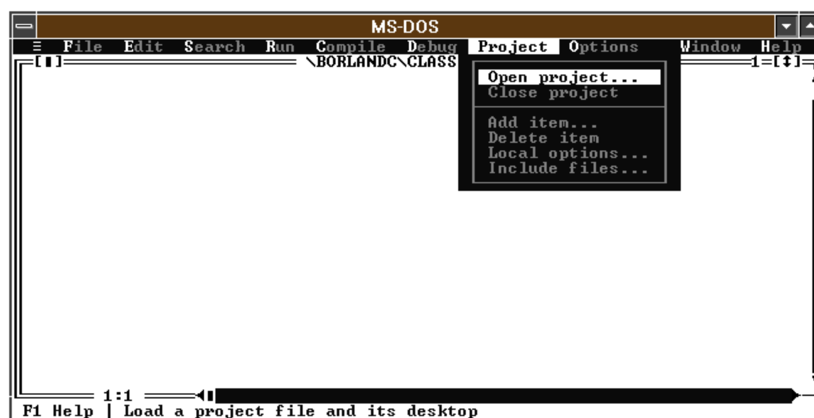
כדי להגדיר ל-Borland C איזה קבצי מקור שייכים לפרויקט מסוים, מכינים קובץ פרויקט בעל סיומת prj, שבו נמצאת רשימה של קבצי המקור. בדוגמה שלנו, קובץ הפרויקט proj1.prj יכיל את קבצי המקור file1.c ו-file2.c. את קובץ הפרויקט ניתן להכין באמצעות חלון העריכה. לשם כך, יש לבחור את הפריט **P**roject, בתפריט הראשי (או על ידי הקשת Alt+P בכל עת) ולבחור בו את הפריט **P**roject name. כעת יתקבל חלון קלט, שבו ניתן להקיש את שם קובץ הפרויקט: proj1. לאחר מכן, יש לבחור את הפריט **C**ompile מהתפריט הראשי, ובתפריט **C**ompile המתקבל יש לבחור את האופציה **B**uild all. מנקודה זו ואילך, ימשיך המהדר באופן עצמאי. תחילה, הוא יבצע הידור לקבצים file1.C ו-file2.C ויקבל את הקבצים file1.obj ו-file2.obj. אם ההידור מצליח עבור שני הקבצים, יחל תהליך הקישור. קובץ התוצר ייקרא על שם קובץ הפרויקט, proj1.exe אשר ניתן להריץ אותו.

כעת נציג מהלך מפורט של עשיית פרויקט במערכת בורלנד.

כיצד נבנה פרויקט

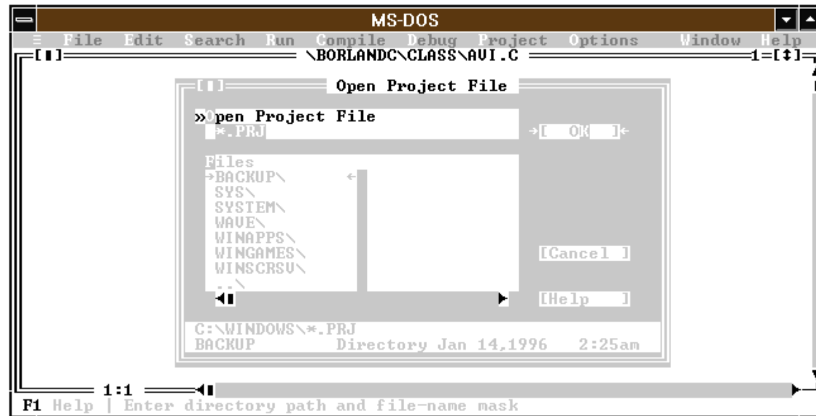
כיצד נבנה פרויקט במערכת Borland C?

1. הצעד הראשון - פתיחת פרויקט חדש.



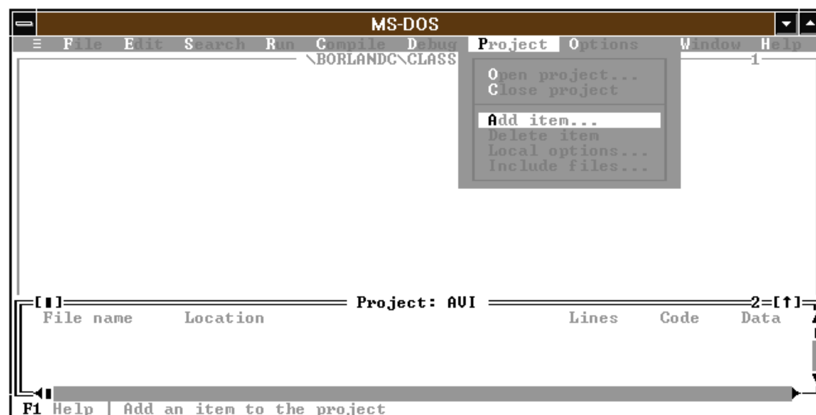
תרשים 41.1: פתיחת פרויקט חדש.

2. נבחר שם לפרויקט עם סיומת ‎*.prj‎.



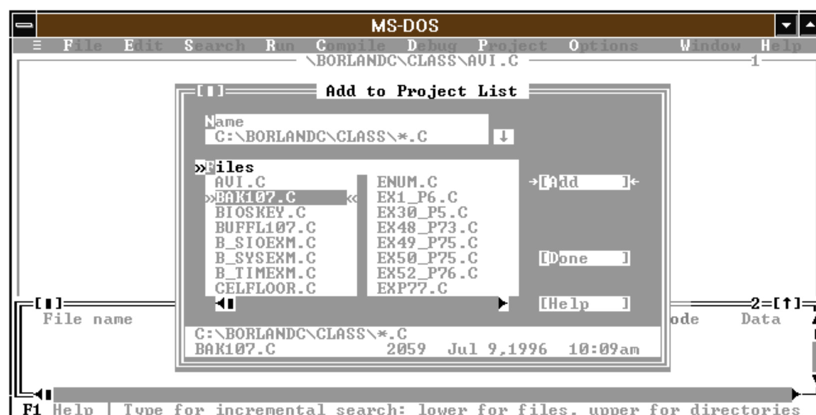
תרשים 41.2: קביעת שם לפרויקט.

3. נקבל חלון פרויקט ונכתוב את שמות הקבצים אשר ישתתפו בפרויקט.



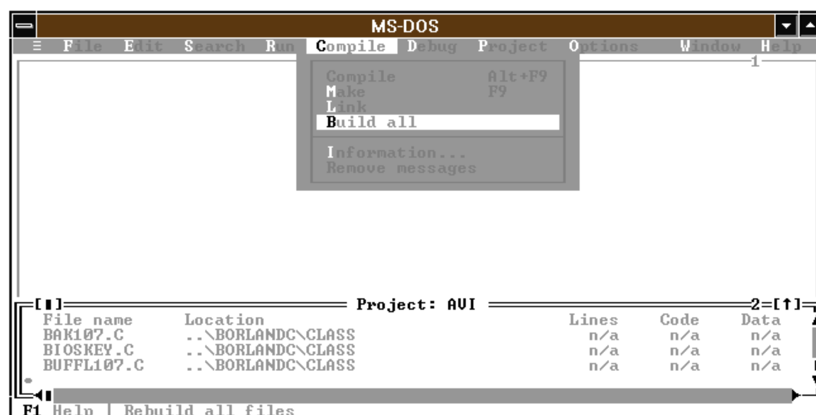
תרשים 41.3: נעבור לסימון הקבצים הדרושים.

4. נקבל תיבת דו-שיח. נמקם את הסמן על שם הקובץ ונבחר את הקבצים המשתתפים. ראה בתרשים 41.4 להלן.



תרשים **41.4**: סימון הקבצים "המשתתפים" בפרויקט.

5. כדי לבנות קובץ ריצה אחד (exe), יש לבחור בפונקציה Build all שבתפריט Compile.



תרשים **41.5**: הוראות ביצוע לבניית הפרויקט.

למעשה, תהליכי ההידור ו/או הקישור לא מבוצעים תמיד. המהדר בודק את התאריך של כל קובץ, ומשווה אותו לתאריך ולשעה של קובץ התוצר, שאמור להיווצר בכל שלב. רק כאשר קובץ המקור **חדש יותר** מקובץ התוצר יבוצע ההידור. לדוגמה, לאחר עשיית הפרויקט נבצע שינוי (בחלון העריכה) לתוכנית file2.C, ומייד לאחר מכן נבצע תהליך עשיית פרויקט. המהדר יבצע את שלב ההידור לקובץ file2.C בלבד, כי על פי התאריך והשעה הוא **חדש יותר** יחסית ל-file2.obj. לא יבוצע כל הידור ל-file1.C, כי הוא לא השתנה מאז ההידור הראשון. לאחר מכן יבוצע שלב הקישור, שכן file2.obj **חדש יותר** מקובץ התוצר proj1.exe. מסיבה זו, חשוב לעדכן את התאריך והשעה במחשב בכל פעם שמתחילים לעבוד (עדכון השעה מתבצע אוטומטית, אם מותקן במחשב כרטיס שעון מגובה סוללה).

מההסבר האחרון ניתן לראות שקיימת מערכת סמיכויות בין הקבצים:

- file1.obj מסתמך על file1.c.
- file2.obj מסתמך על file2.c.
- proj1.exe מסתמך על file1.obj ועל file2.obj.

תוכנית דוגמה לפרויקט

התוכנית הבאה מורכבת משלושה קבצים:

- קובץ ראשי, mainfile.c המכיל את הפונקציה הראשית main, ומפעיל פונקציות שנמצאות בקובץ file_a.c.
- קובץ המכיל פונקציות לחיבור ומכפלה של שני מספרים שלמים, וכן פונקציית הזדהות המדווחת על שם הקובץ ומה הוא יודע לעשות.
- קובץ file_b.c היודע לספר שלוש בדיחות. את הבדיחות בוחרים על-פי מספר הבדיחה. פונקציה זאת מפעילים מהקובץ הראשי mainfile.c.

תהליך חיבור הקבצים ויצירת קובץ הפרויקט הוסבר בסעיף הקודם. התהליך במלואו, על כל שלביו, בוצע על קבצים אלו.

דוגמה:

הקבצים המשתתפים הם:

```
file_a.c
file_a.h
file_b.c
file_b.h
mainfile.c
/*
File            : mainfile.c
File Type       : Source
קובץ ראשי
*/

#include <stdio.h>
#include <conio.h>
/* יש להכליל את ההכרזות על הפונקציות שנמצאות בקבצים אשר בנינו. הכרזות
אלו נועדו לאפשר לנו להשתמש בפונקציות, כדי שהמהדר "יערוך איתם היכרות", שידע
על קיומן. הכרזות אלו מצטרפות להכרזות על פונקציות פנימיות המוגדרות בקובץ זה.
תהליך הבנייה של כל הקבצים לקובץ ריצה אחד מכוון לריכוז כל הפונקציות יחד. */
```

```
#include "file_a.h"
#include "file_b.h"

      /* הכרזות על פונקציות */
void file_a_action(void);
void file_b_action(void);

/* -----------------------------------------------------------
; M A I N - P R O G R A M
 ; -----------------------------------------------------------*/
int main(void)
 {
      clrscr();
      file_a_action();
      file_b_action();
      printf("Press any key");
      getch();
      return(0);
 }
/* -----------------------------------------------------------
; F U N C T I O N S
 ; -----------------------------------------------------------*/
void file_a_action(void)
 {                      /* file_a.c הפעלת הפונקציות בקובץ */
      file_a_i_am();
      printf("add(int 3, int 4) = %d\n", add(3, 4));
      printf("mult(int 3, int 4) = %d\n", mult(3, 4));
 }

void file_b_action(void)
 {                /* file_b.c הפעלת הפונקציות בקובץ */
      file_b_i_am();
      joke_selector(1);
      joke_selector(3);
 }

/*
File            : file_a.c
File Type       : Source
קובץ פונקציות ראשון
*/
```

```
#include <stdio.h>
#include <conio.h>
/* ----------------------------------------------------------
; F U N C T I O N S
 ; --------------------------------------------------------*/
void file_a_i_am(void)
 {
  printf("\nשלום! אני קובץ file_a ואני יודע לחבר ולהכפיל 2 מספרים שלמים\n");
 }

int add(int a, int b)
 {
  return(a + b);
 }

int mult(int a, int b)
 {
  return(a * b);
 }

/*
File            : file_a.h
File Type       : Header
קובץ הכרזות לפונקציות בקובץ file_a.c
*/

void file_a_i_am(void);
int add(int a, int b);
int mult(int a, int b);

/*
File            : file_b.c
File Type       : Source
קובץ פונקציות שני
*/
#include <stdio.h>
#include <conio.h>

void file_b_i_am(void)
 {
  printf("\nשלום! אני קובץ file_b ואני יודע לספר בדיחות\n");
 }
```

```
void joke_selector(int jokeNumber)
 {
  switch (jokeNumber)
      {
        case 1: printf("\nמה אוכלים קניבלים בדיאטה ? גמדים\n");
                          break;
        case 2: printf("\nכמה גרוזינים דרושים בשביל להוריד תמונה מהקיר ?\n
                   עשרה, אחד מחזיק את התמונה ותשעה שוברים ת'קיר\n");
                          break;
      case 3: printf("\nמדוע החבר'ה שחוזרים מגואה לא צריכים אוניברסיטה ?\n
                                                   כל החומר בראש\n");
                          break;
      }
 }

/*
File            : file_b.h
File Type       : Header
file_b.c קובץ הכרזות לפונקציות בקובץ
*/

void file_b_i_am(void);
void joke_selector(int jokeNumber);
```

תוצאות ריצת התוכנית:

```
שלום! אני קובץ file_a ואני יודע לחבר ולהכפיל 2 מספרים שלמים
add(int 3, int 4) = 7
mult(int 3, int 4) = 12

שלום! אני קובץ file_b ואני יודע לספר בדיחות

מה אוכלים קניבלים בדיאטה? גמדים

מדוע החבר'ה שחוזרים מגואה לא צריכים אוניברסיטה?
כל החומר בראש
```

פרק 42

הרצת תוכנית ב־Visual C++

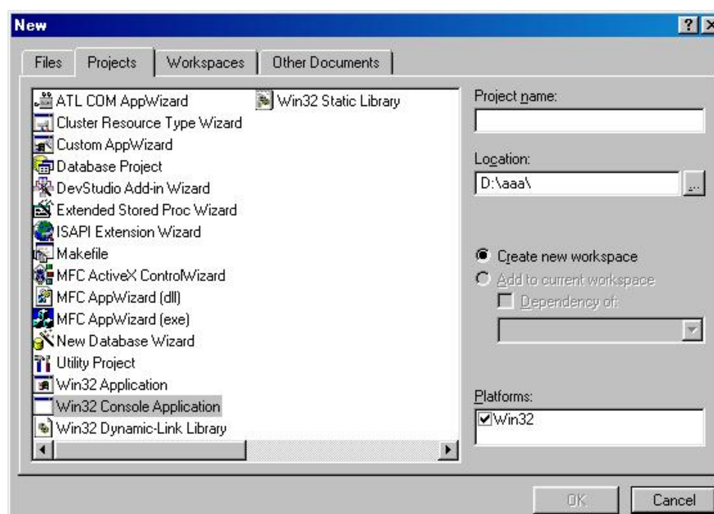
פרק זה נכתב על ידי **אבי בוץ**.

הפרק ניתן כתוספת לספר. אין הוא מתיימר ללמד Visual C++, אלא להראות לך את הדרך כיצד להריץ את התוכניות שבספר או כיצד להתחיל לבנות תוכנית חדשה. ההוראות מתאימות לגירסה 5 ומעלה.

תיאור תהליך הרצת תוכנית חדשה ב – Visual C++

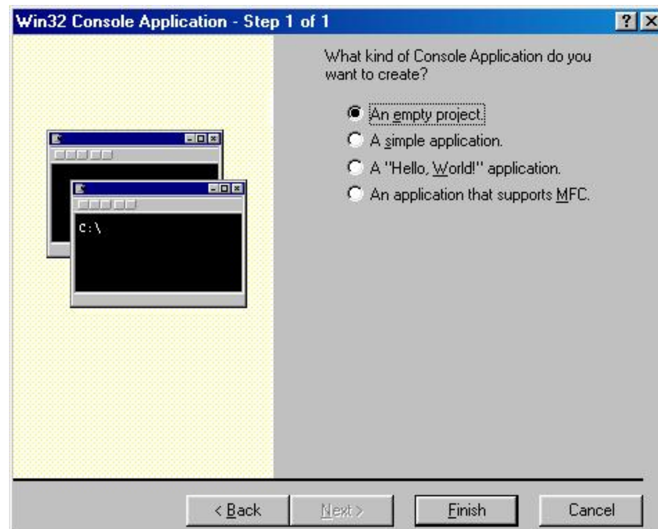
שלב 1

פתח פרויקט מסוג Win32 Console Application, אשר יאפשר להריץ את התוכניות שבספר. תן לפרויקט שם <Project Name> ולחץ על לחצן OK.



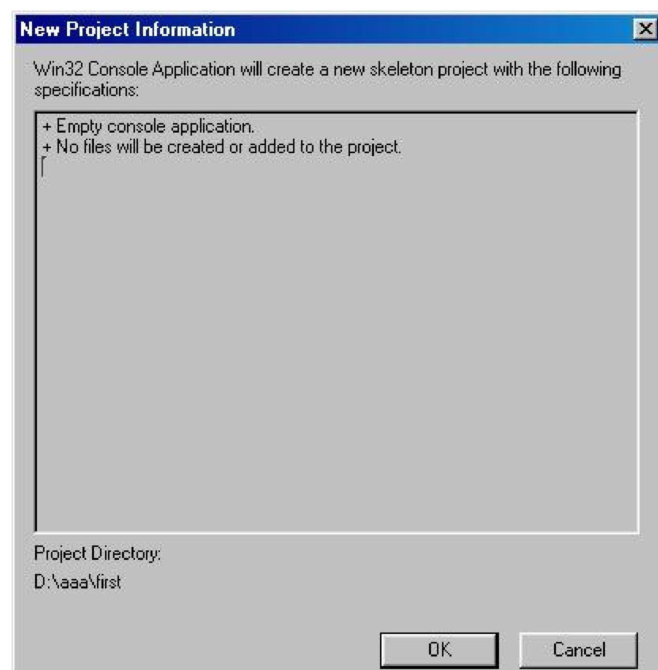
שלב 2

בחר פרויקט ריק <An empty project> ולחץ על לחצן Finish. פרויקט ריק מאפשר ליצור קובץ cpp ריק, אשר בו תכתוב את התוכנית שלך.



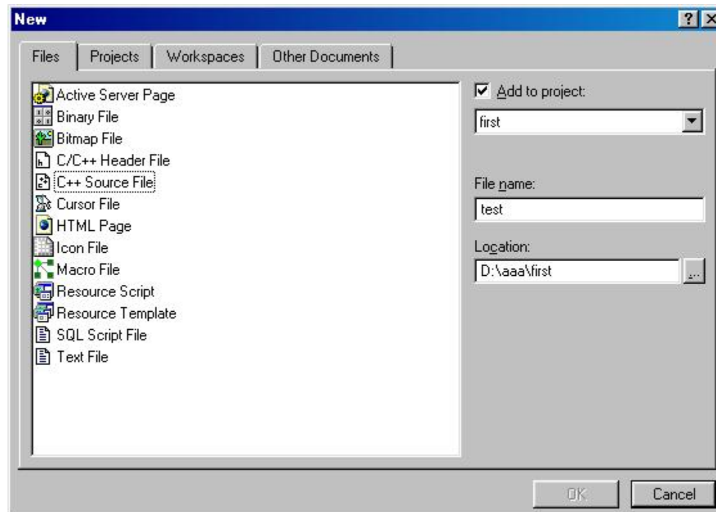
שלב 3

בחלון זה תתבקש לאשר את סוג הפרויקט שבחרת. לחץ על לחצן OK.



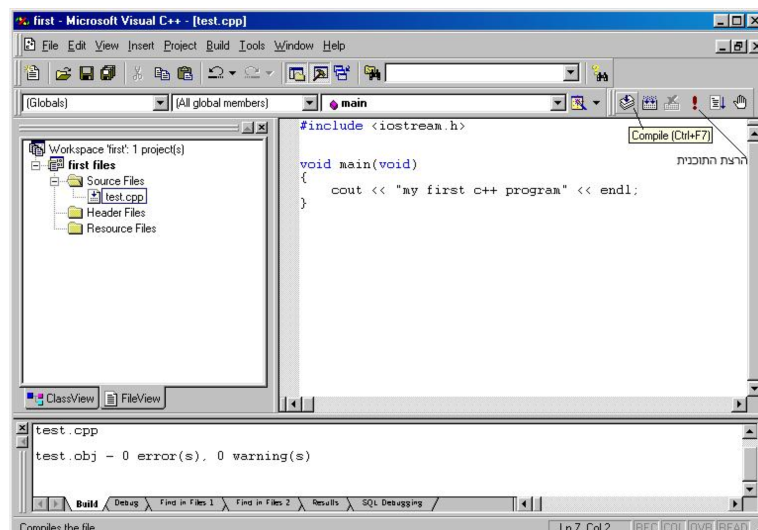
שלב 4

לאחר בחירת הפרויקט יש ליצור קובץ cpp ריק עבור התוכנית. בתפריט File בחר New. ב-Tab בחר Files, וב-Files בחר C++ Source File. תן שם לקובץ File name ולחץ על OK לאישור יצירת הקובץ.



שלב 5

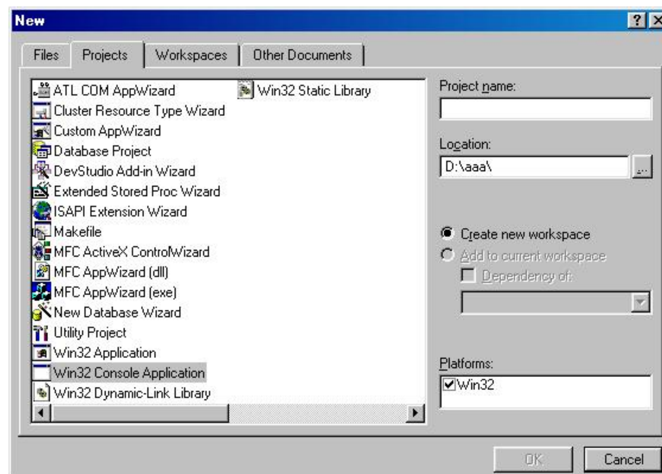
בחלון זה תראה את התוצאה הסופית. ב- Tab בחר FileView כדי לראות את הקובץ שבחרת (test.cpp) כשהוא ריק. לאחר כתיבת התוכנית בדוק שגיאות בעזרת צירוף המקשים Ctrl+F7 ובעזרת צירוף המקשים Ctrl+F5 תוכל להפעיל את התוכנית, כשתהיה כזו.



תיאור תהליך הרצת תוכנית קיימת ב – Visual C++

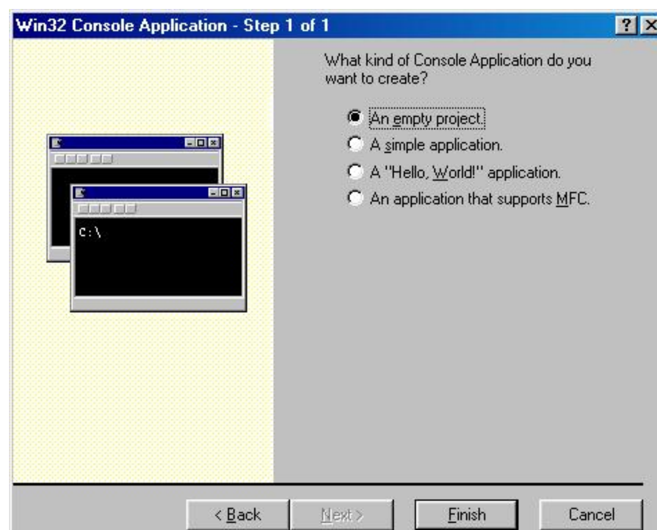
שלב 1

פתח פרויקט מסוג Win32 Console Application, אשר יאפשר להריץ את התוכניות שבספר. תן לפרויקט שם <Project Name> ולחץ על לחצן OK.



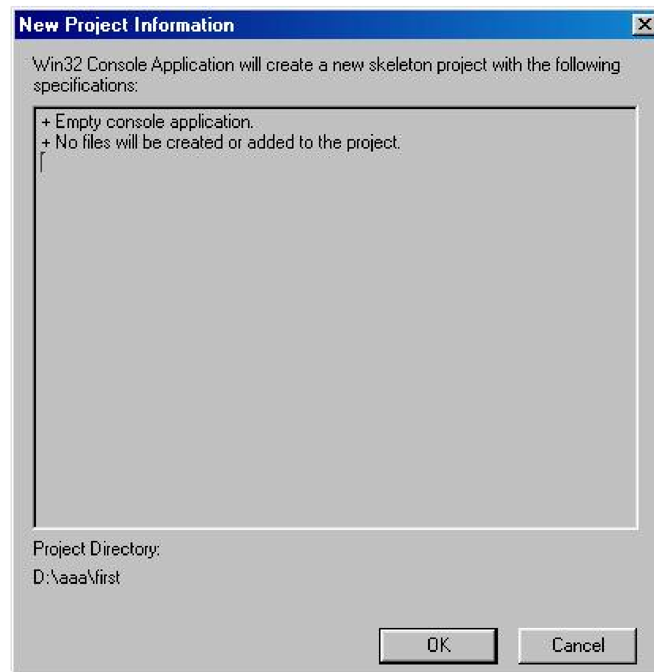
שלב 2

בחר פרויקט ריק <An empty project> ולחץ על לחצן Finish. פרויקט ריק מאפשר ליצור קובץ cpp ריק, אשר בו תכתוב את התוכנית שלך.



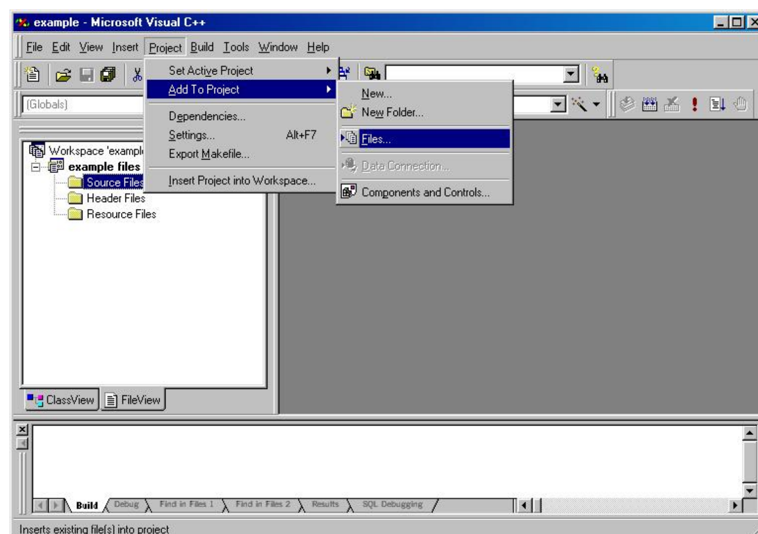
שלב 3

בחלון זה תתבקש לאשר את סוג הפרויקט שבחרת. לחץ על לחצן OK.



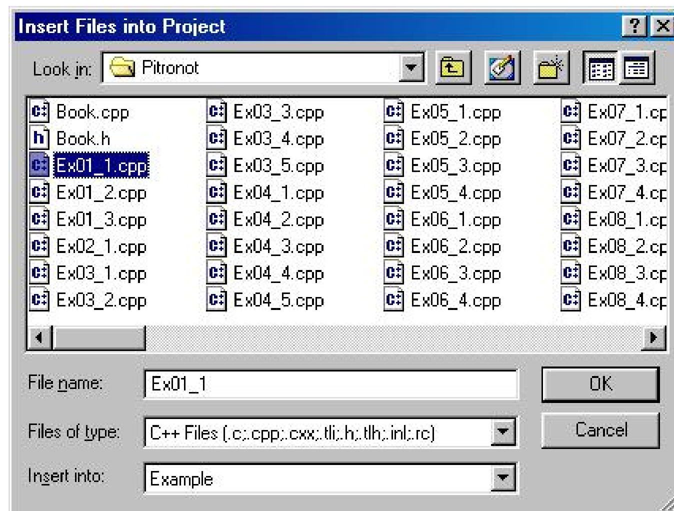
שלב 4

מתפריט Project בחר Add To Project <- Files... כדי להוסיף את הקובץ הקיים.



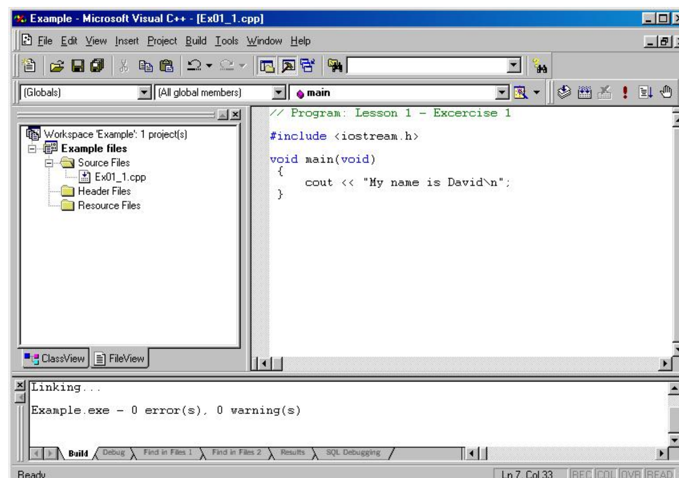
שלב 5

מחלון Insert Files into Project בחר את הקובץ הרצוי ולחץ על לחצן OK.

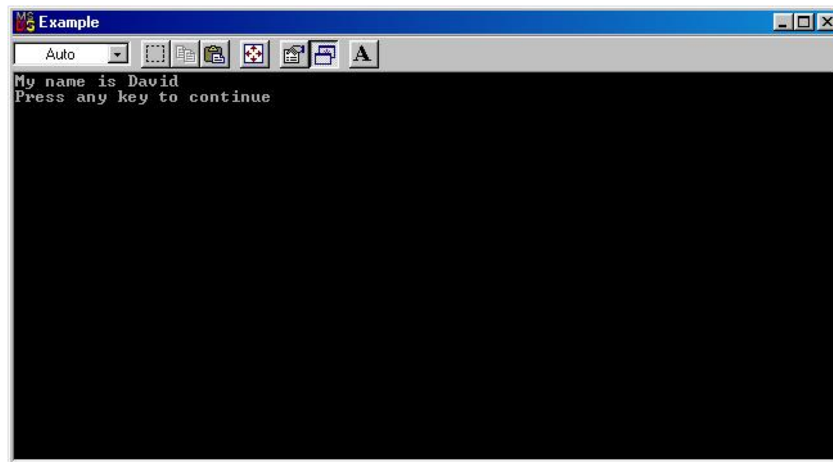


שלב 6

הצג את התוכנית, בדוק אותה (Ctrl+F7) והפעל אותה (Ctrl+F5).



בחלון זה נראה את התוצאה.



נספח

מילון מונחים לעובר מ־C ל־ ++C

כדי להשלים את הספר בחרנו להוסיף מילון מונחים אשר יעזור למתכנתים בשפת C ל"עלות כיתה" ולהכיר את המונחים הנוספים או/ו השונים ממה שמוכרים להם.

מ-B ל-C ומ-C ל- ++C

בשנת 1967 פותחה שפת תכנות שנקראה BCPL, על ידי Martin Richards. Ken Thompson השתמש ביסודות משפה זו בצירוף יסודות משפת B, אותה פיתח, על מנת ליצור את מערכת ההפעלה UNIX, בשנת 1970 במעבדות Bell. באותן מעבדות פיתח Denis Ritchie בשנת 1972, על בסיס שפת B את שפת C. שפת C, שהיתה ידועה כשפה לפיתוח עבור מערכות UNIX, הוסבה במשך השנים לשימוש ברוב המחשבים. בשנת 1983 הוגדרו סטנדרטים בינלאומיים של השפה (ANSI). בשנת 1980 פיתח Bjarne Strousper במעבדות בל את השפה עליה נסוב ספר זה, ++C. השם כולל את אופרטור ההוספה של C, כדי להורות ש- ++C היא גירסה משופרת של C.

נכון להיום, שפת B אינה עוד בשימוש. במקומה שולטת שפת C ו"אחותה" הצעירה, שפת ++C. נספח זה מכוון אל תוכניתני שפת C אשר עוברים לתכנות בשפת ++C. ראוי לציין שרוב התוספות שנעשו בשפה מכוונות לתמיכה ב**תכנות מונחה עצמים** (Object Oriented Programing), ונושא זה מבואר בהרחבה בספר. עם זאת, כמה תכונות נוספו כדי לפשט, להרחיב או לשפר, את סביבת העבודה של התכנות ב-C. מטרת נספח זה להצביע על מספר שינויים במינוח ובפקודות, אשר יקלו על תוכניתני שפת C בהגיעם ללימוד שפת ++C. הנושאים ניתנים כאן בתמצית, מכיון שהם מוסברים בהרחבה בגוף הספר.

מן הראוי לציין שהשפות C ו-++C מתפתחות בגרסאות שונות כל הזמן, ועל כן ייתכן שיימצאו הבדלים שונים, או שהבדלים שאנו מציינים כאן אינם קיימים יותר.

מערכות קלט/פלט מבוססות מחלקה

שפת ++C תומכת בכל מערכות הקלט/פלט של C, אולם בנוסף לכך, היא מגדירה מערכות מבוססות מחלקה. היתרון בשימוש במערכות מבוססות מחלקה של ++C הוא בכך, שהן תומכות בתכנות מונחה עצמים ויכולות לפעול על אובייקטים המוגדרים על ידי המשתמש. התמיכה של ++C במערכות קלט/פלט נמצאת בקובץ הכותר IOSTREM.H. המחלקה ios היא המחלקה הבסיסית שבה משתמשים בדרך כלל לצורכי קלט/פלט ב- ++C. בטבלה הבאה מוצגות מחלקות של ++C והמטרות שלהן :

מחלקה	מטרה
istream	קלט כללי
ostream	פלט כללי
iostream	קלט/פלט כללי
ifstream	קלט לקובץ
ofstream	פלט לקובץ
fstream	קלט/פלט לקובץ
istrstream	קלט מבוסס-מערך
ostrstream	פלט מבוסס-מערך
strstream	פלט/קלט מבוסס מערך

ערוצים מובנים ב- ++C

כאשר תוכנית ++C מתבצעת, נפתחים ארבעה ערוצים מובנים.

ערוץ	משמעות	התקן
cin	ערוץ קלט בסיסי	מקלדת
cout	ערוץ פלט בסיסי	מסך
cerr	ערוץ פלט שגיאה בסיסי	מסך
clog	גרסת מאגר של cerr	מסך

ערוצים חלופיים ב- ++C

הפקודה printf המוכרת מ-C נקראת cout ב- ++C.

הפקודה scanf המוכרת מ-C נקראת cin ב- ++C.

לדוגמה:

```
    printf ("Enter Your name");
    scanf ("%d", &name);
=========================================
    cout << "Enter your name" ;
    cin >> name;
```

יש לשים לב לסימן **קטן מ-** הכפול (>>) שנוסף ב- ++C. זהו **אופרטור הכנסה** (stream insertion operator). הסימן גדול מ- הכפול (<<) הוא **אופרטור ההוצאה** (stream extraction operator). יש לשים לב שסימנים אלה הם בעלי משמעות נוספת ב- ++C: הזחה לשמאל ולימין בהתאמה. המהדר יידע להבחין בין שתי המשמעויות בהתאם לתבנית שבה נמצאים הסימנים. ראה הסבר מפורט על סימנים אלה בספר.

כפי שרואים בדוגמה ב- ++C אין צורך להשתמש במציינים d% ו-&.

אחת המטרות של שינוי זה היא הפיכת התוכניות של ++C לקריאות יותר.

קריאה לפי שם, או על ידי ייחוס

בשפת C קוראים לפונקציה נעשית על פי ערך. שפת ++C מבצעת קריאה ישירה על פי שם, על ידי **ייחוס** (**reference**). כדי לציין שמשתנה הוא משתנה מיוחס, יש להוסיף לפני הטיפוס של המשתנה ב**אב-טיפוס של הפונקציה** (**function prototype**) או **בהצהרת הפונקציה** (**function declaration**) את הסימן &. בקריאת הפונקציה יש לציין את שם המשתנה אשר יועבר על ידי ייחוס. באופן זה ניתן לעדכן את ערך המשתנה המקורי מתוך הפונקציה. משתנים מיוחסים יכולים להיווצר גם בתוך פונקציה.

התבנית הכללית היא:

```
int& parameter_name
```

הגדרות משתנים ב-++C

כל ההגדרות בשפת C חייבות להופיע לפני כל משפטי הרצה. שפת ++C מאפשרת לתוכניתן למקם הגדרות קרוב למקום בתוכנית שבו משתמשים במשתנים (אך כמובן לפני). דבר זה הופך את התוכניות לקריאות יותר.

```
   int gil, int sachar;

   printf ("Enter your age")
   scanf ("%d", &gil);
   printf ("Enter your salary")
   scanf ("%d", &sachar);
   printf("Your salary is ...")
=================================================
   cout << "Enter your age"
   int gil;
   cin >> age;
   cout << "Enter your salary"
   int sal;
   cout << "Your salary is..."
   cout << sal;
```

יש לשים לב שהגדרת המשתנים מופיעה אחרי משפטי הרצה שונים, אך סמוך **ולפני** השימוש במשתנים הכלולים בהגדרה.

כתיבת הערות בתוכנית

כתיבת תוכניות מחייבת לפעמים הוספת הערות בגוף התוכנית, שמטרתן לעזור בהבנתה על ידי תוכניתנים אחרים, שלא כתבו אותה (ולעיתים על ידי התוכניתן עצמו, מכיון שלאחר זמן הוא עצמו עלול לשכוח את משמעות הפעולות או הפונקציות). פעולה שימושית זאת נעשית ב-C באמצעות סימון של קו אלכסוני וכוכבית **בתחילת ההערה ובסופה** (/* גוף ההערה */). הערות ב- ++C מסומנות **בתחילתן בלבד** בשני קווים אלכסוניים בלבד (גוף ההערה //).

לדוגמה:

```
/* this is a comment in C */
// this is a comment in C++

/* this is a comment
that is used
in C programs*/
=========================================
// this is the
// way comments
// are used in c++

int /* this is a comment */ i = 5;
```

שימוש בסגנון זה לכתיבת הערות יכול למנוע היווצרות שגיאות, כמו בזמן ששוכחים לסגור הערה עם סימן קו אלכסוני וכוכבית.

יש לזכור, שאת הסימון להערות בשפת ++C יש לרשום **בתחילת** ההערה **בכל** שורה בה היא מופיעה. מכיון שהמהדרים של ++C מבינים תוכניות הכתובות ב-C, התוכנית תובן גם אם ההערות נכתבו בסגנון של C.

מילים שמורות

32 מילים שמורות מוכרות הן ב- C והן ב- ++C:

auto	break	case	char	const	continue
default	do	double	else	enum	extern
float	for	goto	if	int	long
register	return	short	signed	sizeof	static
struct	switch	typedef	union	unsigned	void
volatile	while				

בשפת ++C יש מילים שמורות **נוספות**. אלו הן המילים אשר בשימוש רק ב- ++C:

asm

שילוב שפת אסמבלי בתוך תוכנית ++C. תבנית שימוש מקובלת:

```
asm instruction;

asm {
   instruction sequence
}
```

catch

פעולה אשר מתבצעת בשילוב עם try ו-throw לטיפול במצבים חריגים בתוכנית: לכידה של המצב החריג.

class

הגדרת מחלקות. תבנית מקובלת:

```
class tag : inheritanc-list {
     //private members by default
protected:
     // private members that may be inherited
public:
     // public members
} object-list;
```

delete

(מחליף את free של C)

משחרר (free) או הורס (destruct) את האובייקט שהוכנס לזיכרון על ידי new (ראה בהמשך). תבנית מקובלת:

```
delete p_variablename;
```

friend

הגדרת פונקציה או מחלקה כחברה של פונקציה או מחלקה אחרת. פונקציה חברה יכולה לגשת לאלמנטים הציבוריים והפרטיים (ראה בהמשך) של המחלקה. תבנית מקובלת:

```
class function_name {
     // ...
public:
     friend void function (int a, float b);
    // ...
};
```

inline

ציון למהדר לשלב את פקודות הפונקציה ברצף הפקודות שבקובץ, ולא כפונקציה נקראת (הגדרה מעין זו יכולה לחסוך בזמן עבודה של התוכנית ובגודלה). תבנית מקובלת:

```
inline void function_name (int  )
{
    //  ...
}
```

new

(מחליף את malloc של C)

הקצאת זיכרון עבור אובייקט. בשפת C היה צורך להגדיר את גודל הזיכרון שרוצים להקצות, ואילו ב- C++ מספיק להוסיף new לפני שם האובייקט כדי לקבל את הקצאת הזיכרון הדרוש ולהחזיר מצביע אל הזיכרון מהסוג המתאים. תבנית מקובלת:

```
p_variablename = new typename;
```

יש הגדרות נלוות נוספות ל- new (ראה בגוף הספר).

operator

השפה מאפשרת שימוש שונה, המוגדר על ידי המשתמש, ברוב האופרטורים הקיימים בשפה (חוץ מ: **. :: .* ?:**).המהדר יפרש את האופרטור לפי ההקשר שלו.הפקודה הזו מגדירה **פונקציה של אופרטור מועמס** (**overloaded operator function**).יש להבחין בין מונח זה לבין המונח **פונקציה מועמסת** (**overloading function**), אשר

מתאר את האפשרות שקיימת ב- C++ להגדיר מספר פונקציות עם אותו השם, אבל עם פרמטרים שונים.

תבנית מקובלת לאלמנט של מחלקה:

```
return_type class-name::operator #(parameter-list) {
   // ...
}
```

תבנית מקובלת לאלמנט שאינו של מחלקה:

```
return_type operator #(parameter--list) {
   // ...
}
```

private

מציין הגישה של האלמנט הפרטי של המחלקה, הנגיש לפונקציות חברות מטיפוס של האלמנטים הפרטיים. תבנית כללית:

```
class tag {
   // ...
private: // make private
   // private elements
};
```

תבנית להגדרה של אלמנט פרטי כמוריש:

```
class tag : private base-class { //...
```

protected

מציין הגישה המוגן הוא הגדרה לאלמנט מורחב של private, הנגיש גם לפונקציות נגזרות. תבנית מקובלת:

```
class tag{
   // ...
protected: // make protected
   //  protected elemants
};
```

public

מציין הגישה הציבורי הוא הגדרה לאלמנט מחלקה הנגיש לכל פונקציה.
תבנית מקובלת:

```
class tag {
   // private elements
public: // make public
   // public elements
};
```

כאשר משתמשים בו כמציין מוריש, התבנית הכללית היא:

```
class tag : public base-class { // ...
```

template

המילה השמורה template (תבנית) מאפשרת ליצור תבניות של מחלקות ופונקציות כלליות. בפונקציות הכלליות מוגדר סוג הנתון כפרמטר, לכן פונקציה אחת יכולה לשרת מספר טיפוסים שונים של נתונים. מוגדר בה סדרה של אופרטורים כלליים אשר מיושמים לטיפוסים שונים של נתונים. תבנית מקובלת:

```
template <class data-typr>
type func-name(parameter list)
{
   // body of function
}
```

this

זהו מצביע (pointer) לאובייקט, אשר פונקצית חברה קוראת לו.

throw

פעולה המתבצעת בשילוב עם try ו-catch לטיפול במצבים חריגים בתוכנית: הכרזת מצב חריג.

try

פעולה המתבצעת בשילוב עם throw ו-catch לטיפול במצבים חריגים בתוכנית: בחינת מצב חריג.

virtual

מציין הפונקציה virtual מגדיר פונקציה וירטואלית, שהיא טיפוס של חברה במחלקה בסיסית. התוכנית תשתמש במחלקה הבסיסית הווירטואלית במידה והיא לא תידרס על ידי מחלקה נגזרת. תבנית מקובלת:

```
virtual return_type fname(param-list) = 0;
```

מפתח סימנים מיוחדים

רשימת טבלאות

אינדקס עברי

ב

ד

ה

ז

ט

י

ל

מ

נ

ס

ע

פ

אינדקס לועזי

לשם נוחות הקריאה התחלנו את האינדקס הלועזי מסוף הספר. כלומר, סופו בעמוד הבא ותחילתו כמו בספר באנגלית.

אינדקס לועזי

D

E

F

R

S

V